

A Computational Logic Handbook

Second Edition

Academic Press
International Series in Formal Methods

Series Editor: Michael G. Hinchey

Previously published title:

Teaching and Learning Formal Methods, 1996

This book is printed on acid-free paper.

Copyright © 1998 by Academic Press

All Rights Reserved.

No part of this publication may be reproduced or transmitted in any form or by any means, electronic or mechanical, including photocopy, recording, or any information storage and retrieval system, without permission in writing from the publisher.

Academic Press
525 B Street, Suite 1900, San Diego, California 92101-4495 USA
<http://www.apnet.com>

Academic Press Limited
24-28 Oval Road, London NW1 7DX, UK
<http://www.hbuk.co.uk/ap/>

ISBN 0-12-122955-6

A catalogue record for this book is available from the British Library.

Symbolics and Zetalisp are trademarks of Symbolics, Inc. Sun, Sun 3/60, and Sparc are trademarks of Sun Microsystems, Inc. Lucid and Lucid Common Lisp are trademarks of Lucid, Inc. Unix is a trademark of AT&T Information Systems, Inc. Allegro CL is a trademark of Franz, Inc. Macintosh is a registered trademark of Apple Computer, Inc. Scribe is a registered trademark licensed exclusively to Cygnet Publishing Technologies, Inc.

Printed in Great Britain by Hartnolls Ltd, Bodmin, Cornwall

98 99 00 01 02 03 EB 9 8 7 6 5 4 3 2 1

A Computational Logic Handbook

Second Edition

Robert S. Boyer
and
J Strother Moore

ACADEMIC PRESS

Harcourt Brace & Company, Publishers
San Diego, London, Boston, New York, Sydney, Tokyo, Toronto

To John McCarthy,

for the idea of Lisp programs and
the idea of proving theorems about them

Series Foreword

As early as 1949, computer pioneers realized that writing a program that executed as planned was no simple task. Turing even saw the need to address the issues of program correctness and termination, reflecting the work of von Neumann and Goldstine, and foretelling groundbreaking work of John McCarthy, Edsger Dijkstra, Bob Floyd and Tony Hoare, and Sue Owicki and David Gries two and three decades later.

The term *formal methods* refers to the application of mathematical techniques for the specification, analysis, design, implementation and subsequent maintenance of complex computer software and hardware. These techniques have proven themselves, when correctly and appropriately applied, to result in systems of the highest quality, which are well documented, more easy to maintain, and which promote software reuse.

With emerging legislation, and increasing emphasis in standards and university curricula, formal methods are set to become even more important in system development. The *Academic Press International Series in Formal Methods* aims to present results from the cutting edge of formal methods research and practice.

Books in the series will address the use of formal methods in both hardware and software development, and the role of formal methods in the development of safety-critical and real-time systems, hardware-software co-design, testing, simulation and prototyping, software quality assurance, software reuse, security, and many other areas. The series aims to include both basic and advanced textbooks, monographs, reference books and collections of high quality papers which will address managerial issues, the development process, requirements

engineering, tool support, method integration, metrics, reverse engineering, and issues relating to formal methods education.

It is our aim that, in due course, the *Academic Press International Series in Formal Methods* will provide a rich source of information for students, teachers, academic researchers and industrial practitioners alike. And I hope that, for many, it will be a first port-of-call for relevant texts and literature.

Professor Mike Hinchey

Preface to the First Edition

This book is a user's guide to a computational logic. A "computational logic" is a mathematical logic that is both oriented towards discussion of computation and mechanized so that proofs can be checked by computation. The computational logic discussed in this handbook is that developed by Boyer and Moore.

This handbook contains a precise and complete description of our logic and a detailed reference guide to the associated mechanical theorem proving system. In addition, the handbook includes a primer for the logic as a functional programming language, an introduction to proofs in the logic, a primer for the mechanical theorem prover, stylistic advice on how to use the logic and theorem prover effectively, and many examples.

The logic was last described completely in our book *A Computational Logic*, [24], published in 1979. The main purpose of the book was to describe in detail how the theorem prover worked, its organization, proof techniques, heuristics, etc. One measure of the success of the book is that we know of three independent successful efforts to construct the theorem prover from the book.

In the eight years since *A Computational Logic* was published, the logic and the theorem prover have changed. On two occasions we changed the logic, both times concerned with the problem of axiomatizing an interpreter for the logic as a function in the logic but motivated by different applications. The first attempt was motivated by the desire to permit metatheoretic extensibility: the efficient use of functions in the logic as new proof procedures once they have been proved sound. This work was completely described in [25] and resulted in minor changes to the formalization of literal atoms, the introduction of **QUOTE** notation, and axioms defining a weak interpreter for a fragment of the logic.

The second attempt was motivated by the desire to permit bounded quantification and the introduction of partial recursive functions. This work was described in [30]. It scrapped the previously added axioms for an interpreter and added axioms defining a more powerful one.

In both [25] and [30] we described accompanying changes in the theorem prover itself, most notably in the heuristics for expanding recursive functions (where we no longer always expand a function when its controlling arguments are explicit constants) and new heuristics for simplifying calls of the interpreter on constants representing terms.

There have been two truly important changes to the theorem prover since 1979, neither of which has had to do with additions to the logic. One was the integration of a linear arithmetic decision procedure. The other was the addition of a rather primitive facility permitting the user to give hints to the theorem prover.

During the period 1980-1984 we integrated into the rewrite-driven simplifier a linear arithmetic decision procedure. The goal—which was not entirely achieved—was to free the heuristic component of the simplifier from having to deduce the consequences of a decidable fragment of its theory, permitting the axioms for that fragment to be ignored. In the case of linear arithmetic this meant we could eliminate from the simplifier’s view such troublesome facts as the transitivity of less than and the monotonicity of the Peano addition function. However, to achieve the intended goal it was necessary to make the decision procedure serve the simplifier in all the roles that the omitted axioms had previously. This was very difficult because of the complexity of the simplifier. We eventually adopted a decision procedure for the rationals which gave good performance and coverage in the integer case but which is not complete for the integer case. The resulting “linear arithmetic package” has proved to be very robust and useful. The entire history of our experiment and the final arrangement of heuristics and the decision procedure is completely described in [31].

The second important change was the addition of a primitive hint facility. The two most commonly used hints can be paraphrased as “consider the following instance of the previously proved theorem **n**” and “do not consider using the previously proved theorem **m**.” Prior to the implementation of the hint facility, the user could guide the system to a proof—as we did for the prime factorization theorem in [24]—by the proper statement of lemmas to be proved before the main theorem. The hint facility just made it less difficult to lead the theorem prover when that was necessary. Because our hint facility contributed no new ideas to the literature of automatic theorem proving we have not documented it previously.

The most important changes have occurred not in the logic or the code but in our understanding and use of them. The past eight years have seen the heavy use of the logic and theorem prover. They have been pushed into successful

applications far beyond those we reported in *A Computational Logic*. The most impressive number theoretic result proved in 1979 was the existence and uniqueness of prime factorizations; it is now Gauss's law of quadratic reciprocity. The most impressive metamathematical result was the soundness and completeness of a propositional calculus decision procedure; it is now Gödel's incompleteness theorem. These results are not isolated peaks on a plain but just the highest ones in ranges explored with the system. Among others are Fermat's theorem and Wilson's theorem, the unsolvability of the halting problem, the tautology theorem, and the Church-Rosser theorem. In program verification it is now hardly meaningful to choose a single peak. In 1979 it was the correctness of the Boyer-Moore fast string searching algorithm; among the many we are now proud of are the invertibility of the RSA public key encryption algorithm, the equivalence of sequential and parallel executions of a bitonic sort algorithm, the correctness of a gate-level design of a microprocessor, the correctness of a simple multitasking operating system, and the correctness of a compiler and link-assembler for a nontrivial programming language. An annotated list of references to the important results proved by the system is given on page 5.

The improvements to the logic and theorem prover, especially linear arithmetic and the hint facility, have certainly helped make these conquests possible. But perhaps the most important change since the publication of *A Computational Logic* was that in 1981 we moved from SRI International, where we were involved exclusively in research, to the University of Texas at Austin. Our research home at the University of Texas was the Institute for Computing Science and Computer Applications. However, as professors in the Department of Computer Sciences, we teach. In 1981 we began teaching a graduate course, now called *Recursion and Induction*, on how to prove theorems in our logic, and we initiated a weekly logic seminar attended by graduate students, other faculty members, and logicians from local research organizations. These efforts dramatically increased the number of people familiar with our work. In addition, we began using the theorem prover to check the proofs of theorems we wanted to present in class (e.g., the unsolvability of the halting problem).

Almost all of the results we are now most proud of were carried out under the direction of former students who saw our mechanically checked proofs of the simpler theorems (unsolvability or RSA) and came to us believing they could lead the expeditions to the harder theorems. Almost invariably we thought they were wrong—they simply didn't understand how obstinate the theorem prover is, how weak the logic is, how many hacks and kludges there are inside. And almost invariably we were right about their naivete but wrong about their talent and persistence. Time and time again our students came forth with impressive results. We haven't revised our opinion of our system—it is obstinate, the logic is weak, and there are many hacks. But we have nothing but praise for the

graduate students who have used our system with such success: David Russinoff, Warren Hunt, Natarajan Shankar, C.-H. Huang, Bill Bevier, Bill Young, David Goldschlag, Jimi Crawford, and Michael Vose.

In summary, there have been comparatively minor changes to the logic and theorem prover since the publication of *A Computational Logic*, but there have been dramatic increases in both the difficulty of the problems to which the system has been applied and the number of people who use the system. We believe that the theorem prover as it stands today (except for the provisions for hints) is adequately described by the combination of [24, 25, 31, 30]. We describe the hint facility here and do not otherwise attempt the thorough documentation of our techniques. What was lacking, in our view, was a thorough treatment of how to use the logic and the theorem prover. That is the main objective of the current work.

To make this volume self-contained we present a complete description of the current logic and as much information about how the system works as is necessary to describe how to use it. That said, however, we think of this book as a continuation of *A Computational Logic* rather than a replacement for it.

The decision to document the logic and system now should not be taken as a declaration that it is “fixed.” In particular, we still have serious misgivings about the pragmatics of our new interpreter for the logic, **V&C\$**, and the bounded quantifier, **FOR**. We will, undoubtedly, either improve the system’s handling of these functions or we will replace them entirely with new and hopefully better formal constructs. Our decision to document the current system stems in part from our perceived scientific duty to tell the community exactly what we are doing and in part from the realization that the user community has grown sufficiently large that written documentation will save us the time we otherwise spend telling people how to use the system.

As previously noted, it is to our user community that we owe our biggest debt. The contributions of the graduate students of the University of Texas, especially those that we supervised, cannot be overemphasized. In particular, we acknowledge the contributions of Bill Bevier, S.C. Chou, Ernie Cohen, Jimi Crawford, David Goldschlag, C.H. Huang, Warren Hunt, Myung Kim, David Russinoff, Natarajan Shankar, and Bill Young. In addition, we have profited enormously from our association with Hans Kamp, Chris Lengauer, Norman Martin, John Nagle, Carl Pixley, and Bill Schelter. Topher Cooper has the distinction of being the only person to have ever found an unsoundness in a released version of our system.¹

Many people have contributed to this handbook, including all of the users named above. We would like to especially thank Natarajan Shankar, who

¹This was still true in 1997, as far as we know, when the second edition went to press.

helped write part of Chapter 4 and some of the proofs in Chapter 5. In addition, many errors in the handbook were found by students who took our courses. We especially thank Tom Verhoeff and Sam Dooley, who read the handbook very carefully.

Finally, special note must be made of the contributions of Matt Kaufmann. Matt has used the theorem prover to do several interesting proofs, including the correctness of a unification algorithm and the correctness of a Towers of Hanoi solution. In addition, Matt undertook the careful scrutiny of the preliminary version of the code implementing bounded quantification and partial functions. His months of study uncovered several subtle bugs and led to many clarifying comments and helpful suggestions. In addition, he went over the handbook with a fine-tooth comb.

We also most gratefully acknowledge the support of our colleagues at the Institute for Computing Science and Computer Applications, especially Don Good and Sandy Olmstead who created and maintained at the Institute a creative and relaxed research atmosphere with excellent computing facilities. In 1986 we moved our entire verification research group off campus and established Computational Logic, Inc.

Notwithstanding the contributions of all our friends and supporters, we would like to make clear that ours is a very large and complicated system that was written entirely by the two of us. Not a single line of Lisp in our system was written by a third party. Consequently, every bug in it is ours alone. Soundness is the most important property of a theorem prover, and we urge any user who finds such a bug to report it to us at once.

The development of our logic and theorem prover has been an ongoing effort for the last 15 years. During that period we have received financial support from many sources. Our work has been supported for over a decade by the National Science Foundation and the Office of Naval Research. Of the many different grants and contracts involved we list only the latest: NSF Grant DCR-8202943, NSF Grant DCR81-22039, and ONR Contract N00014-81-K-0634. We are especially grateful to NSF, ONR, and our technical monitors there, Tom Keenan, Bob Grafton, and Ralph Wachter, for years of steady support and encouragement. We have received additional support over the years from the following sources, listed chronologically: Science Research Council (now the Science and Engineering Research Council) of the United Kingdom, Xerox, SRI International, NASA, Air Force Office of Scientific Research, Digital Equipment Corporation, the University of Texas at Austin, the Venture Research Unit of British Petroleum, Ltd., IBM, National Computer Security Center, and Space and Naval Warfare Systems Command.

Since 1984 the major part of our funding has come from the Defense Advanced Research Projects Agency. At UT we were supported by ARPA Order Number 5246. At Computational Logic, Inc., we are supported by ARPA Order

Numbers 9151 and 6082. The research described here was supported in part by the U.S. Government. The views and conclusions contained in this book are those of the authors and should not be interpreted as representing the official policies, either expressed or implied, of the Defense Advanced Research Projects Agency or the U.S. Government.

The first draft of this handbook was written entirely by Moore and required slightly over a year to write. During that time his wife, Miren, and their children, Lisa, Nicolas, Jonathan, and Natalie, made many sacrifices so that he could work with few interruptions. We are especially grateful to them for their understanding and patience.

Preface to the Second Edition

Nine years have elapsed since the publication of the first edition of *A Computational Logic Handbook* and the theorem proving system it described, Nqthm. During that period, several significant new features were added to the Nqthm system. These features were implemented in “patch” files and combinations of these files were in use in verification projects at Computational Logic, Inc. (CLI) and the University of Texas (UT). Some of these projects have produced significant new Nqthm proofs. Unfortunately, because of the use of the patch files, it was not always easy to communicate to outsiders what had been proved or how. We therefore decided to rationalize the situation by releasing an extended Nqthm. We also took the opportunity to correct the reported “bugs” in Nqthm (though none involved soundness). The result is called Nqthm-1992. In trying to document the changes we came to the conclusion that the most coherent approach we could take would be to produce a second edition of this book.

Nqthm-1992 includes new events for introducing constrained function symbols, inferring theorems via functional instantiation, and defining, enabling and disabling theories. The new system supports **COND**, **CASE**, **LET**, **LIST*** and “backquote” syntax. Performance is improved markedly when large sets of names are disabled and when large case structures arise. More support is offered for compiling and loading library files and for processing event files to produce “endorsed” data bases or for “skimming” event files to construct a fraudulent data base for experimental purposes. Some less noticeable changes were also made. The new release includes approximately fifteen megabytes of example Nqthm event files.

It would be nice to say that any sequence of events acceptable to the old Nqthm is acceptable to Nqthm-1992, but this is not true. There is one syntactic incompatibility: Nqthm-1992 does not allow duplicate occurrences of hints while Nqthm does. However, because of some heuristic changes it is possible for the new system to fail to find proofs that the old system finds. In our many megabytes of example files we found a few examples of this. All of the failures in those files were caused by the fact that the new “fast clausifier” sometimes produces clauses in which the literals occur in a different order than in Nqthm. This can affect the subsequent instantiation of free variables or expose loops in rewrite rules that were present but not exposed before. Matt Kaufmann found two more instances in which heuristic “improvements” caused old scripts not to replay. One failure was caused by a change in Nqthm’s preferred ordering for terms, which is used to decide whether a permutative rewrite rule should be applied; in the new ordering, explicit values come before non-explicit values, regardless of their lexicographic size; because the ordering has always been sensitive to the number of variables in the two terms, this change is very rarely felt because explicit values contain no variables and were thus “naturally” given priority over all non-explicit values except (the rare) irreducible ground expressions such as terms involving undefined constants. The second failure noted by Kaufmann was due to the fact that Nqthm-1992 can apply metafunctions to tame generalizations of “nontame” terms and hence metatheoretic simplifiers may see more use under Nqthm-1992 than under the older version. These exceptions notwithstanding, the most common experience with Nqthm-1992 is that old scripts replay without any trouble. However, users should be alert to the possibility that a script will have to be changed to make it replay.

Library files produced by the old Nqthm are not compatible with Nqthm-1992. We recommend that the new system be used to rebuild old libraries from the event lists.

Installation instructions for Nqthm-1992 are almost unchanged from those for the previously released version of Nqthm.

In many cases, our former colleagues at CLI first implemented the features we are now adding. We are indebted to Bill Bevier for his invention of the notion of “theories” and their implementation, Matt Wilding for his implementation of “fast disabling,” and Bishop Brock for his “fast clausifier.” Matt Kaufmann, David Goldschlag and Bill Bevier collaborated with us in the development of the logical basis for and the design of the “constrain” and “functionally instantiate” events. Bill Schelter has rendered invaluable service to us by maintaining and improving the performance of Austin Kyoto Common Lisp (AKCL), now known as Gnu Common Lisp (GCL).

The contributions of Matt Kaufmann deserve a special note. Matt has written an interactive enhancement to Nqthm, called “Pc-Nqthm” [75, 76, 82, 21]. It is

an interactive *proof checker* for the Nqthm logic (hence the name, *Pc-Nqthm*). As such it provides low-level commands such as **induct**, **expand**, **=** (substitute), etc. It also provides high-level commands that employ the Nqthm theorem prover, such as **prove** (a subgoal). In addition, *macro commands* provide a capability similar to that provided by tactics in HOL [64], and *Pc-Nqthm* provides support for an extension of the logic that includes first-order quantification. *Pc-Nqthm* is integrated with Nqthm so that one can freely mix standard Nqthm-style proof development with interactive proof checking. Some theorems proved using *Pc-Nqthm* include the correctness of a high-level language compiler [154], several Unity programs [60, 61, 62], and correctness proofs for parametrized hardware modules [139]. Enquiries about *Pc-Nqthm* may be sent to nqthm-users-request@cs.utexas.edu.

The ability to use the first-order quantifiers “for all” and “there exists” can be beneficial for Nqthm users, especially for the purpose of writing clear specifications. Included in the *Pc-Nqthm* system is a facility called **DEFN-SK** [83] that provides this capability. (The “**SK**” in “**DEFN-SK**” stands for “Skolem,” since **DEFN-SK** uses the technique of *Skolemization* to define functions whose definition bodies contain quantifiers.) In [83] Matt has used this enhancement in verifying some relatively deep theorems: Koenig’s Tree Lemma, the infinite exponent-2 Ramsey’s Theorem, and the Schroeder-Bernstein Theorem. Others have made use of **DEFN-SK** in their work on proving properties of computing systems. For example, David Goldschlag [61] has found **DEFN-SK** useful in formalizing the UNITY logic inside the Nqthm logic, and Yuan Yu [33, 159, 35] has used **DEFN-SK** in his verification of MC68020 machine code for the Berkeley C String library. Although **DEFN-SK** is part of *Pc-Nqthm*, it can be loaded directly on top of Nqthm as a standalone enhancement.

Matt’s creation of *Pc-Nqthm* has contributed in many ways to the success of Nqthm. For example, it has made Nqthm available to people who preferred a proof-checking environment. It has even helped users who prefer Nqthm-style proof development: using *Pc-Nqthm* to test a certain rewriting strategy often precedes the “automation” of that strategy via **REWRITE** rules. But perhaps the biggest contribution of *Pc-Nqthm* has been to make Matt Kaufmann intimately familiar with our code for Nqthm. His knowledge of Nqthm, combined with his apparently boundless energy and enthusiasm, made him point man in the mission to improve Nqthm: our former colleagues at CLI would generally talk to Matt before they spoke to us about proposed changes. Matt undertook the responsibility of maintaining the myriad “Nqthm patch files” that floated around the CLI Nqthm community. Matt made valuable contributions to the design and implementation of all of the features noted above, always keeping soundness uppermost in his mind. His contributions to and scrutiny of proposed changes has made it much easier for us to accept the changes. Without Matt’s

efforts and enthusiasm, it is unlikely Nqthm-1992 would have contained any of the new features.

After obtaining permission from the implementors above, we integrated the new ideas into our source code, often changing the implementation to bring it wholly within our style. Thus, despite our gratitude to our colleagues for their ideas and code, we remain entirely responsible for the correctness of Nqthm.

This release of Nqthm includes approximately fifteen megabytes of Nqthm script files. These files, when processed by Nqthm-1992, produce over 300 megabytes of proof descriptions and endorsed data bases. These files illustrate how Nqthm's logic can be used to formalize a wide variety of problems and how Nqthm can be led to deep proofs. A wide variety of topics are covered: number theory, logic, metamathematics, algorithms, hardware design, machine code programs, linkers, assemblers, compilers, multiprocessing, concurrency, asynchronous communications, and game playing. The magnitude of our debt to our user community is illustrated by the fact that the scripts assembled here are an order of magnitude larger than the implementation of Nqthm. The following people have contributed scripts to this release: Bill Bevier, Bishop Brock, Alex Bronstein, John Cowles, Art Flatau, Mike Green, Warren Hunt, Matt Kaufmann, Ken Kunen, Stan Letovsky, Misao Nagayama, Paolo Pechiari, David Russinoff, Natarajan Shankar, Sakthi Subramanian, Carolyn Talcott, Matt Wilding, Bill Young, and Yuan Yu.

This most recent release of Nqthm also contains an “infix printer,” an optional utility one may use to reformat Nqthm event scripts into more conventional mathematical notation. The infix printer translates an Nqthm event file into a LaTeX or Scribe source file, which may then be processed by the appropriate document preparation system. For example, the event

```
(prove-lemma boolean-sample (rewrite)
  (implies (and (lessp a b)
                (lessp b (length c)))
    (zerop (difference a (length c)))))
```

may be printed as

THEOREM: boolean-sample
 $a < b \wedge b < \text{length}(c) \rightarrow a - \text{length}(c) \equiv 0$

The infix printer was initially written by Boyer to produce LaTeX output and was then extended by Mike Smith to produce Scribe output. See the installation guide for more details.

We would like to thank Alan Boxer, Warren Hunt, Matt Kaufmann, Donovan Kolbly, Bill Legato, Sylvan Pinsky, Larry Smith, and Morgan Witthoft for pointing out errors in the first edition of this book and earlier drafts of this edition. For proof reading diligence above and beyond the call of duty, we

would like to thank Simon Read. This edition would be a less suitable manual had it not been for his many suggestions; and it would have been more suitable still had we had the patience to implement more of his suggestions.

We also thank our past sponsors for their many years of very substantial support, especially Bill Scherlis and Steve Squires of the Defense Advanced Research Projects Agency; Ralph Wachter of the Office of Naval Research; and Norm Glick, Terry Ireland, and Bill Legato at the National Security Agency.

This work was supported in part at Computational Logic, Inc., by the Defense Advanced Research Projects Agency, ARPA Order 7406 and the Office of Naval Research, Contract Number N00014-91-C-0130. The views and conclusions contained in this document are those of the authors and should not be interpreted as representing the official policies, either expressed or implied, of Computational Logic, Inc., the sponsoring organizations, or the U.S. Government.

Since the early 1990s Kaufmann and Moore have been working on a new theorem prover, called “A Computational Logic for Applicative Common Lisp,” or “ACL2.” ACL2 is similar to Nqthm but supports an applicative subset of Common Lisp as its logic. Furthermore, this new theorem prover is written almost entirely in that subset of Common Lisp. See <http://www.cs.utexas.edu/users/moore/acl2> or [84, 39].

Although our research prospered for approximately a decade at Computational Logic, Inc., its doors closed on August 1, 1997, solely because of drastic changes in federal research support. Moore has now returned to being a professor in the Computer Sciences Department of the University of Texas at Austin, where Boyer had maintained his teaching position. Most of our former colleagues now work at software or hardware firms in the Austin area.

Computational Logic, Inc., was a delightful place to do research. We wish to express our gratitude to the many who made it such a wonderful place, including to its presidents Don Good and Mike Smith; to the administrative staff, including Sandy Olmstead, Laura Tice, and Nancy Wagner; to the computing staff, including Ron Olphie and Charles Sandel; and most of all to our many former colleagues who applied Nqthm in so many insightful ways and also contributed to its development.

Biographical sketches of the authors may be found in [18] and [103]. Their email addresses are boyer@cs.utexas.edu and moore@cs.utexas.edu, and their home pages are <http://www.cs.utexas.edu/users/boyer> and <http://www.cs.utexas.edu/users/moore>. Correspondence may be addressed to the authors at the Department of Computer Sciences, University of Texas at Austin, Austin, Texas 78712.

Part I

The Logic

1 Introduction

By a “computational logic,” we mean a logic that is both oriented towards discussion of computation and mechanized so that proofs can be checked by computation. We are the authors of one such logic and its associated mechanical theorem prover. This book explains how to use the logic and the theorem prover.

The book is divided into two parts. In Part I we discuss the logic, without regard for its mechanization. In Part II we discuss its mechanization. Part I answers the questions “what are the axioms and rules of inference?” and “how does one use the logic to state and prove theorems?” Part II answers the question “how does one use the Boyer-Moore theorem prover to prove theorems?”

Do not be put off by the size of the documentation. Much of the verbiage here is devoted to spoon-fed introductions and primers, examples, and stylistic advice on how to use the logic and its implementation effectively.

Two chapters have been designed as reference guides.

- Chapter 4 is the technical heart of Part I and is a detailed description of the syntax, axioms, rules of inference and extension principles of the logic. In addition to defining the logic, we carefully define many metalinguistic and metatheoretical notions used in the description of the logic.
- Chapter 12 is the technical heart of Part II and is a reference guide to the theorem prover. It is organized so that you can read only the parts relevant to a given command or concept and cross-referenced so that you can track down unfamiliar notions.

Having tried to allay your fears of the book itself, we now give a brief overview of the logic and the theorem prover and what has been done with them. Then we give an overview of this book.

1.1. A Summary of the Logic and Theorem Prover

Technically the logic is a quantifier-free, first-order logic resembling Pure Lisp [95, 96, 97, 98, 99, 100]. Its axioms and rules are obtained from the propositional calculus with equality and function symbols by adding (a) axioms characterizing certain basic function symbols, (b) three “extension principles,” with which one can add new axioms to the logic to introduce “new” inductively defined “data types,” recursive functions, and constrained functions and (c) mathematical induction on the ordinals up to ε_0 as a rule of inference. However, among the “certain basic functions” axiomatized is an interpreter for the partial recursive functions which then permits the introduction of concepts with the same essential properties as bounded quantifiers and such higher-order features as schemators and functional objects. Furthermore, a derived rule of inference permits the derivation of a new theorem from an old one by the substitution of certain new function symbols for old function symbols—a seemingly higher-order act that provides much of the power and convenience of functional parameters.

The logic is mechanized by a collection of Common Lisp [131, 132] programs that permit the user to axiomatize inductively constructed data types, define recursive functions, and prove theorems about them. This collection of programs is frequently referred to as “the Boyer-Moore theorem prover,” although the program that implements the theorem prover itself is only one—albeit the largest—of many programs provided in the system. The prover is written in what is sometimes called the Bledsoe [11, 12] style in automated reasoning.

The theorem prover’s behavior on a given problem is determined by a data base of rules. The rules are derived by the system from the axioms, definitions, and theorems submitted by the user. In particular, each time a new theorem is proved it is converted into rule form and stored in the data base. When new theorems are submitted the currently “enabled” rules determine how certain parts of the theorem prover behave. In this way the user can lead the machine to the proofs of exceedingly deep theorems by presenting it with an appropriate graduated sequence of lemmas. In such applications the theorem prover resembles a sophisticated proof checker more than an automatic theorem prover.

While the user’s participation in the proof discovery process is crucial to the construction of deep proofs, the user assumes no responsibility for the correctness of the proofs constructed. That is the burden of the theorem prover—or,

more accurately, the burden of its authors. However, to use the system effectively it is necessary to understand the kinds of rules used by the system, how each class of rule affects the system's behavior, and how rules are generated from formulas.

The logic may be viewed as an applicative programming language. It is possible to define functions in the logic and execute them with reasonable efficiency on concrete data, without ever *proving* anything about them. A simple execution environment is provided as part of the system.

The logic and theorem prover have been used to formalize and prove many interesting theorems in mathematics, metamathematics, theory of computation, and computer science. We give an annotated list of selected applications in the next section. The theorem prover is written in Common Lisp in accordance with *Common Lisp The Language* [131, 132]. The source code is more than 900,000 bytes long.

1.2. Some Interesting Applications

First-time users of the logic and theorem prover are frequently frustrated by the weakness of the language and the inability of the theorem prover to see the obvious. A commonly expressed thought is “You can’t do anything with this system!”

Below is an annotated list of selected references to theorems proved by the system. We include it for two reasons. First, just knowing that some people—and not merely the authors—have used the logic and theorem prover to do interesting things is often enough encouragement to get you started. Second, if you are stuck and cannot see how to proceed, perhaps the way will be clearer if you obtain and read the relevant-sounding work mentioned below. In fact, as noted in Chapter 14, the standard distribution of the system comes with many megabytes of example event files. Many of the results cited below are contained in those files, and we indicate which examples are in which files. Because the actual file names are long we refer to them here by number. By looking up the file number in the list starting on page 423 you can find the file name, the file’s author, a bibliographic reference if available, and a brief description of the file. Because not every example file described in Chapter 14 is discussed in this introductory chapter, a reader wishing to know whether a topic has already received Nqthm scrutiny is well advised also to examine the full list in Chapter 14.

You should understand, in view of the foregoing remarks on the role of the user, that—for the deep theorems at least—the theorem prover did not so much *discover* proofs as *check* proofs sketched by the user. We therefore wish to acknowledge, collectively, the users whose work we cite below: Bill Bevier,

Dominique Borri one, Jon Bright, Bishop Brock, Alex Bronstein, Miren Carranza, Luc Claesen, Jeff Cook, John Cowles, Ben DiVito, Art Flatau, Steve German, Paul Gloess, David Goldschlag, Mike Green, C.-H. Huang, Warren Hunt, Matt Kaufmann, Ken Kunen, Chris Lengauer, Stan Letovsky, Bill McCune, Misao Nagayama, Paolo Pecchiari, Laurence Pierre, David Russinoff, John Ramsdell, Simon Read, Natarajan Shankar, Ann Siebert, Larry Smith, Mike Smith, Sakthi Subramanian, Carolyn Talcott, Diederik Verkest, Debbie Weber-Wulff, Matt Wilding, Bill Young, and Yuan Yu. See the references (both bibliographic and to the example files documented on page 423) for individual credits.

- **Elementary List Processing:** Many elementary theorems about list processing are discussed among the examples in [24]. The appendix includes theorems proved about such concepts as concatenation, membership, permuting (including reversing and sorting), and tree exploration. In addition, many users have studied elementary list processing functions while learning the system. See example files 13, 55, 42, 45, and 47 for some representative examples, and also [49]. A formalization of a card trick having to do with the results of a random shuffle of a deck of cards initially arranged to alternate in color is an interesting exercise in list processing and is presented in example file 1. Alternative ways to decompose sequences and the use of our shell principle are presented in example file 25. A model of a simple data base against which read and write transactions can occur is studied in file 41. Finally, in Appendix I (which is also example file 10) we describe the formalization of the Nqthm syntax by means of a parser defined in the logic. The parser maps lists of ASCII character codes to the quotations of formal terms and, in addition, deals with all of our abbreviation conventions such as the quote and backquote notation.
- **Elementary Number Theory:** Euclid’s theorem and the existence and uniqueness of prime factorizations are proved in [24] (see **EUCLID**, **PRIME-FACTORIZATION-EXISTENCE**, and **PRIME-FACTORIZATION-UNIQUENESS** in example file 13). A second version of the prime factorization theorem, one not requiring the introduction of auxiliary functions like **GCD**, is presented in file 52. A version of the pigeon hole principle, Fermat’s theorem, and the invertibility of the RSA public key encryption algorithm (see [117]) are proved in [27] (see file 15). Several versions of Ackermann’s function are studied in files 11 and 50. Libraries of useful rewrite rules about the naturals and integers may be found in files 59 and 58. Sums of factorial expressions are studied in file 24. Some facts about Fibonacci numbers, in particular properties of various sums and Matijasevich’s lemma, are proved in files 5 and 57. Two ver-

sions of Ramsey's theorem have been formalized in Nqthm: the exponent 2 case [79] and the infinite case [83] (see files 43 and 46). A Paris-Harrington Ramsey theorem is proved in [89] (see file 53). Wilson's theorem is proved in [120] (see file 20). Finally, Gauss's law of quadratic reciprocity has been checked; see [121] and also file 9.

- **Metamathematics:** The soundness and completeness of a decision procedure for propositional calculus, similar to the Wang algorithm (see [144]), is proved in [24] (see **TAUTOLOGY-CHECKER-IS-SOUND** and **TAUTOLOGY-CHECKER-IS-COMPLETE** in example file 13). The soundness of an arithmetic simplifier for the logic is proved in [25] (see **CORRECTNESS-OF-CANCEL** in file 13). The existence of nonprimitive-recursive functions is demonstrated in file 12. The Turing completeness of Pure Lisp is proved in [28] (see file 18). The recursive unsolvability of the halting problem for Pure Lisp is proved in [29] (see file 19). A mechanically checked proof of the correctness of a generalization algorithm that operates in the presence of free variables is shown in [80] (see file 39). Koenig's tree lemma is proved in [83] (see file 40). The tautology theorem, i.e., that every tautology has a proof in Shoenfield's formal system, is proved in [126, 127] (see file 65). The Church-Rosser theorem is proved in [128, 127] (see file 63). Gödel's incompleteness theorem is proved in [127, 129] (see file 64). That induction up to ω^2 is provable in Primitive Recursive Arithmetic and that the Nqthm logic is not constructive are both proved in [90] (see file 51). Theorems relating the SECD machine to the Lambda Calculus are presented in [114]. A proof checker coded in the Nqthm logic that has been used to check proof objects produced by the Otter automated reasoning system may be found in [101].
- **Partial Functions:** The mechanically checked proofs of the termination properties of several partial functions are given or sketched in [30] (see example file 14). In particular, we discuss the nontermination of the recurrence $(\mathbf{RUS\ X}) \Leftarrow (\mathbf{ADD1\ (RUS\ X)})$, the fact that a certain recurrence terminates iff the argument is a "proper" list ending in **NIL**, the totality of the 91-function, the totality of the function defined by

```
(RV X)
  ←
  (IF (AND (LISTP X) (LISTP (CDR X)))
      (CONS (CAR (RV (CDR X)))
            (RV (CONS
                  (CAR X)
                  (RV (CDR (RV (CDR X)))))))
      X)
```

and the fact that **RV** reverses its argument. In [50] a generalization of the 91-function is treated. In [102] we prove that the Takeuchi function terminates (see file 21). An alternative approach to partial functions is illustrated in file 44.

- **Bounded Quantifiers and Higher-Order Functions:** Many theorems have been proved about the “higher-order” function **FOR** which takes expressions among its arguments and provides a form of bounded quantification. Among these theorems are classic “schematic” quantifier manipulation theorems such as

$$\sum_{i=0}^n g(i) + h(i) = \sum_{i=0}^n g(i) + \sum_{i=0}^n h(i);$$

In addition, these schematic theorems have been used in the proofs of other theorems requiring quantifiers to state, such as the binomial theorem:

$$(a+b)^n = \sum_{i=0}^n \binom{n}{i} a^i b^{n-i};$$

The proofs of many of these theorems and the techniques used are explained in [30] (see example files 14 and 3). In [19] we describe how our derived rule of functional instantiation can be used to prove theorems about “mapping functions,” recursive schemas, quantifiers, and other concepts usually dealt with in a richer logic (see file 7). In file 38 a method for proving permutation-independence of list processing functions is developed.

- **Communication Protocols:** Safety properties of two transport protocols, the Stenning protocol and the “NanoTCP” protocol, are proved in [52]. The verification of a clock synchronization algorithm is presented in [156]. A model of communications between two independently clocked digital devices is presented in [105]. The model is then used to prove that a biphasic mark protocol using cells of size 18 bits can reliably send arbitrarily long messages between two processors provided the ratio of clock rates of the two processors is within 5.5% of unity. See example file 2.
- **Elementary Language Processing Exercises:** Perhaps the most common problem formalized with Nqthm is the semantics of programming languages or computational paradigms. See page 104. This is typically done initially at the operational level via an interpreter defined as an Nqthm function. Programs, treated as quoted constants, can then be analyzed directly. For some applications, we abstract away from the operational semantics to higher-level proof rules—after establishing the validity of those rules with respect to the operational semantics. This methodology is followed in many of the most complicated applications of Nqthm (enumerated below).

Unfortunately, the complexity of the systems formalized obscures the simplicity of the basic methodology. We therefore recommend that the novice start with simple examples of the approach. In the example file 16 we define operationally the semantics of an assembly-like language. Without further ado we then prove the correctness of several programs in that language. We then show the “functional approach” in which programs are modeled by recursive functions that “do the same thing” to the machine state. We then develop the “inductive assertion” approach and show how proving certain “verification conditions” allows us to conclude facts about the operational semantics. In our standard benchmark of definitions and theorems is a collection that defines another simple assembly language, including “jump” and “move to memory” instructions, and proves the correctness of a program that iteratively computes the sum of the integers from 0 to n . This language differs from the first primarily by the fact that the instructions are fetched from the same memory that is being modified by the execution of the program (see **CORRECTNESS-OF-INTERPRETED-SIGMA** in file 13). Another common endeavor with Nqthm is studying the relations between two formally modeled computational paradigms. The best documented elementary example is in [24], where we define a simple assembly language for a stack machine and a simple expression language, develop an optimizing compiler that maps the expression language to the assembly language, and present a correctness proof for the compiler (see **CORRECTNESS-OF-OPTIMIZING-COMPILER** in file 13). A similar exercise, intended as a challenge for beginners, is given in file 37. The file 48 illustrates the map from functional notation to reverse Polish. As previously noted, Appendix I (and file 10) exhibits a recursive descent parser for the Nqthm syntax. See [59, 145, 146] for other work on theorems about parsing in Nqthm.

- **Concurrent Algorithms:** A mechanized theory of “simple” sorting networks and a proof of the equivalence of sequential and parallel executions of an insertion sort program are described in [92]. A more general treatment of sorting networks and an equivalence proof for a bitonic sort are given in [66]. A proof of the optimality of a given transformation for introducing concurrency into sorting networks is described in [94]. See also [93]. A formalization of Misra and Chandy’s Unity language [47] is described in [60] along with the proofs of several Unity programs. [108] describes a mechanization of a proof originally done informally by Manna and Pnueli establishing the correctness of a mutual exclusion algorithm. See example files 67 and 68.

- **Fortran Programs:** A verification condition generator for a subset of ANSI Fortran 66 and 77 is presented in [26]. The formal definitions of the Nqthm function symbols used in the paper may be found in example file 32. The paper describes the correctness proof of a Fortran implementation of the Boyer-Moore fast string searching algorithm (see file 33). A correctness proof for a Fortran implementation of an integer square root algorithm based on Newton's method is described in [20] (see file 34). The proof of a linear time majority vote algorithm in Fortran is given in [32] (see file 35).
- **Real Time Control:** A simple real time control problem is considered in [20]. The paper presents a recursive definition of a "simulator" for a simple physical system—a vehicle attempting to navigate a straight-line course in a varying crosswind. Two theorems are proved about the simulated vehicle: the vehicle does not wander outside of a certain corridor if the wind changes "smoothly," and the vehicle homes to the proper course if the wind stays steady for a certain amount of time (see example file 4). Theorems about fuzzy controllers are presented in [45, 46]. In [151] may be found the verification of several real time programs for a verified microprocessor. Example file 61 contains a proof of the optimality of an earliest-deadline-first scheduler on any set of periodic tasks. Games represent another arena in which the formalization of control problems may be studied. For example, the problem of implementing a tic-tac-toe playing program may be thought of as trying to build a control program that interacts with a hostile environment. The formal specification of such interaction is challenging. For example, just as a naive specification of sorting may admit the "solution" of returning the empty sequence, a naive specification of the interaction in a game may admit the "solution" of moving the opponent's pieces or not accurately displaying the positions of the pieces. These issues have analogues in the formal specification of control problems, where it is implicit that the control program cannot change the environment except in certain ways. In example file 17 we present a script that formalizes the challenge of building a tic-tac-toe playing program that never cheats and never loses, we define a function that we then prove satisfies the specification, and we refine the function into the recursive expression of a simple number-crunching loop. In file 60 a similar exercise is carried out for the game of Nim. Nim is a game played with piles of sticks. Two opponents alternate taking at least one stick from exactly one pile and the winner is the player who takes the last stick. In file 60 an Nqthm function is shown to implement a strategy that wins provided the initial game state satisfies a certain constraint. In file 30 a Piton program is shown to correctly implement this strategy. In file 66 is a treatment of the well-known

theorem concerning the impossibility of tiling a “mutilated” checkerboard.

- Hardware Verification:** The correctness of a ripple carry adder is given in [73, 70]. The adder is a recursively defined function which maps a pair of bit vectors and an input carry bit to a bit vector and an output carry bit. The theorem establishes that the natural number interpretation of the output is the Peano sum of the natural number interpretations of the inputs, with appropriate consideration of the carry flags. An analogous result is proved for twos-complement integer arithmetic. The recursive description of the circuit can be used to generate an adder of arbitrary width. A 16-bit wide version is shown. Propagate-generate and conditional-sum adders have also been proved correct. Also in [73, 70] is the correctness proof of the combinational logic for a 16-bit wide arithmetic logical unit providing the standard operations on bit vectors, natural numbers, and integers. The main contribution of [73, 70] is a recursively described, microcoded cpu, called the FM8501, which is proved to implement correctly an instruction set defined by a high-level interpreter. The Nqthm events for design, specification and verification of the FM8501 are contained in example file 36. The design was later extended to produce the FM8502, a 32-bit version of the processor. See [72, 71]. More recently, a fragment of a commercial hardware description language (NDL by LSI Logic, Inc.) was formalized in the logic and has been used to describe the design of a new microprocessor, the FM9001 [74, 37]. The semantics of its intended machine code were defined and mechanically proved to be implemented by the design. The FM9001 microprocessor has been fabricated. The hardware description language, the design, and the machine code semantics were formalized within the Nqthm logic, and the proof that the gate-level design correctly implements the machine code semantics was checked by Nqthm. The machine code semantics (the so-called “behavioral level” specification of the FM9001), the gate level description, and the proof that the gates correctly implement the behavioral level may all be found in example file 29. Another extensive hardware verification effort based on Nqthm is that reported in [41, 40, 42, 43]. In that work, “string-functional semantics” for circuits are formalized and used to prove the correctness of many designs, with special emphasis on pipelined synchronous circuits. See example file 23. A mechanically checked proof of a Byzantine agreement algorithm maybe be found in [10], and the verification of hardware implementing Byzantine agreement is presented in [104]. Other reports on hardware verification projects that have made use of Nqthm include [17, 16, 125, 15, 110, 109, 111, 112, 13, 14, 113, 57] and [115, 116, 123, 124, 139, 140, 141, 143, 2, 142].

- **Machine Code:** A formalization of the machine code for the Motorola MC68020 microprocessor is described in [159] and more briefly in [33, 35]. The user visible state consists of 8 data registers, 8 address registers, a 32-bit program counter, an 8-bit condition code register (which includes condition codes for carry, overflow, zero, negative, and extend), and memory. Semantics are formalized for approximately 80 instruction opcodes and all eighteen addressing modes. Breaking the instructions down into the ten categories of the MC68020 user's manual [107], the model formalizes all 9 of the data movement instructions, 27 of the 28 integer arithmetic instructions (**CMP2** is excluded), all 9 of the logical operations, all 9 of the shift and rotate instructions, all 4 of the bit manipulation instructions, all 8 of the bit field instructions, none of the binary coded decimal instructions, 11 of the 13 program control instructions (**CALLM** and **RTM** are excluded), 5 of the 21 system control instructions, and none of the multiprocessor instructions. The formal specification is an operational model involving about 600 Nqthm function definitions. It occupies about 80 pages. See example file 83. An extensive library of rules about these definitions was also developed; see file 84. To demonstrate the viability of the approach, several interesting MC68020 machine code programs have been verified from the formal semantics. These programs are obtained from high-level language programs written in C or Ada and then compiled with commercially available compilers targeting the MC68020. The Gnu C compiler and the Verdex Ada compiler were used. Among the programs so verified are Euclid's gcd program (from a C source program, see file 76), Hoare's *in situ* Quick Sort (from a slightly modified version of the C source in [86], see file 86), binary search (from a C source program, see file 72), integer square root (from an Ada source program, see file 80) and 21 of the 22 programs in the widely used Berkeley C string library, see file 73. See [159] for details.
- **Operating System:** A small multitasking operating system kernel written in the machine language of a uni-processor von Neumann computer is developed and proved correct in [3]. See also [5, 6]. The kernel is proved to implement a fixed number of conceptually distributed communicating processes. In addition to implementing processes, the kernel provides the following verified services: process scheduling, error handling, message passing, and an interface to asynchronous devices. The operating system consists of about 300 executable machine instructions. The listing of the system requires 620 lines, including assembler directives. The system, including its tables and save areas, occupies about 3K words on the host machine (see example file 22). Formalization of parts of the Mach Kernel is discussed in [8, 9].

- **Language Implementation:** The correctness of the implementation of a nontrivial programming language on a microprocessor is described in [106]. The language, called Piton, is a stack-based assembly level language providing seven abstract data types, execute-only programs, global arrays, and many other high-level features. Piton is defined via an interpreter written as a function in the logic. Piton is implemented via a compiler and a link-assembler written as functions in the logic. The first version of the implementation produced machine code for the FM8502 described above. The implementation was proved correct in the sense that the results of non-erroneous Piton computations are correctly produced by executing FM8502 on the binary image produced by the compiler and link-assembler. When FM9001 was produced, Piton was reimplemented for it and the FM8502 proof was “ported” to the FM9001. The definition of Piton, its implementation on the FM9001, and the proof of its correctness may be found in example file 31. Also in [106] we describe a Piton program for doing base 2^{32} “big number addition” and its correctness proof. (See file 28.) In related work, a compiler that maps a higher-level language, Micro-Gypsy, into Piton is described in [154]. The combination of FM8502, Piton, and Micro-Gypsy represents what is called the “CLI Short Stack” and is described in [7]. Dynamic storage allocation is the primary concern in example files 26 and 27. Those files contain a definition and proof of correctness for a compiler from a subset of the Nqthm language to Piton. The subset includes **CONS** and thus the runtime system provided in Piton by the compiler must deal with storage allocation. See [53]. Another Nqthm-based endeavor in garbage collection is [122]. Semantics for Middle Gypsy 2.05, a Pascal-like language, are presented in the Nqthm logic in [63]. Semantics for a subset of the C programming language, one expressed in the Nqthm logic, and the verification of C programs and object code produced from C programs are described in [48, 135, 136]. A sorting certifier is verified in [36]. Floating point arithmetic is considered in [147].

In Appendix III we present a brief comparison of the complexities of several of these theorems and proofs.

1.3. The Organization of this Handbook

Chapter 1 is this introduction.

The logic is presented twice, once in Chapter 2 in tutorial form and again in Chapter 4 in the precise, legalistic English of the logician. This is our attempt to overcome a well-known problem in the presentation of technical material: should the presentation be oriented toward the novice—who wishes to understand the subject but is also desperate for examples, motivation, and explanation of the implications of the various “design decisions”—or should it be oriented toward the knowledgeable reader—who understands the main ideas but wants a succinct and precise presentation to use as a reference guide?

In Chapter 2 we provide an introduction to the logic as a programming language. We familiarize you with our syntax for terms, such as the variable symbol X and the function application $(PLUS\ X\ Y)$, and for constants, such as the “literal atom” constant $'ABC$ and the list constant $'((A\ .\ 7)(B\ .\ 1))$. We describe the behavior of the primitive functions and data types. We illustrate the use of the primitive data types to represent conventional data structures such as lists, tables, trees, and terms. We explain how to add new data types to the logic. We explain the principle of recursive definition and why we have imposed the various restrictions on it. We illustrate some simple recursive definitions and then show the results they produce when evaluated on constants. We illustrate the representation of the ordinals up to ϵ_0 . We discuss the general purpose interpreter for the logic, and we discuss the provisions for bounded quantification. Because we do not exhibit the axioms or rules of inference actually defining the logic in this chapter, we do not deal with the notion of proof here.

In Chapter 3 we discuss the problem of using the logic to formalize problems. This is always a major stumbling block in the path of new users: how to convert the question “is this program (algorithm, function, system, idea) correct?” into the question “is this formula a theorem of the logic?” There are really two problems here: how to use mathematics to formalize everyday concepts in computer science and how to use this logic in particular. The first problem is hard because there is a major philosophical gap between the user’s personal intentions and desires and any formal rendering of them. But many users of the logic wade into this “specification problem” without fully understanding how treacherous it is. We try in this chapter simply to illustrate several specification problems by formalizing some informal specifications for stacks, queues, and a sort function. Then we turn to the specific formalization problems raised by our logic. For example, we discuss the absence of quantification and infinite objects, and we discuss various techniques for formalizing nondeterminism and language semantics.

In Chapter 4 we describe the syntax, axioms, and rules of inference precisely. To be sufficiently precise when explaining our abbreviation conventions and the terms used to describe axiom schemas and rules of inference, we also define in this chapter a host of metalinguistic and metatheoretical notions, e.g., what we mean by the “explosion” of an ASCII character sequence, what we mean by a “type restriction” in the declaration of a new data type, and what we mean by the “governing” terms in a definition. We might have segregated these metanotions into a preliminary chapter, thus drastically reducing the size of the precise description of the logic, but we have decided to define our technical terms as close as possible to their first use in the hopes that by seeing them used immediately the reader will better grasp the concepts. We have carefully labelled each paragraph of the chapter as to whether it concerns terminology, abbreviation conventions, axioms, etc.

Most readers will find Chapter 4 hard going unless they have already read the primer or are already familiar with our logic. Nevertheless, Chapter 4 is one of the two main parts of this handbook. It is absolutely essential that you master the material here if you wish to use the logic correctly. How can the logic be used incorrectly if the machine takes responsibility for the soundness of the proofs? The answer is simple: the machine guarantees that the formulas it pronounces “theorems” in fact follow from the axioms and definitions. But we offer no assurance that those formulas *mean* what you want them to mean (that’s the formalization problem). Unless you really understand the formal logic you will not be able to capture, with definitions and theorems, the concepts and relationships you will be grappling with when you apply the logic.

In Chapter 5 we illustrate how theorems are proved in the logic by carefully explaining many hand constructed proofs, starting at the level of propositional calculus and quickly ascending to inductive proofs about simple recursive functions. Several new rules of inference are derived, including so-called “structural induction.” Essentially this chapter bridges the gap between the sense of “proof” defined by the logic and the sense of “proof” supported by the mechanical theorem prover. The theorem prover’s style of proof is more closely “proof” in the mathematician’s sense than the formal logician’s sense. In particular, the machine does not print out a sequence of formulas, each of which is either an axiom or follows from previous ones by a named rule of inference. The machine takes much larger steps and uses many derived rules of inference (such as decision procedures). The foundation of our trust in the theorem prover is the privately held conviction that every “proof” it produces can, in principle, be converted into a formal proof. It is not necessary that you read Chapter 5 if you accept the theoretical equivalence of these two senses of proof.

Chapter 6 briefly explains the basic proof techniques implemented in our theorem prover and how they are organized. We explain the details of the proof techniques later. Most of this chapter is devoted to a mechanically produced

proof of a simple arithmetic theorem and then a move-by-move analysis of it to acquaint you with the style of the theorem prover's techniques and output.

The theorem prover, *per se*, is just a part of the mechanization of the logic. In addition to proving theorems, the user must be able to carry out other logical acts, such as defining functions and adding new data types. Moreover, there is the key problem of the system's data base of facts which largely determines how well the system performs on problems of practical interest. Finally, there are the mundane but essential features of any computing system: how to get it started, how to stop it, how to save your work, how to find out what is happening. These issues are discussed in Chapter 7. We also summarize the commonly used commands, discuss error handling and typein, and conclude with a few cautionary words about possible confusions arising from the similarity of the logic to Lisp.

Chapter 8 is just a demonstration of the theorem prover. We show it proving several theorems about the simple list processing function **REVERSE**. These proofs show the theorem prover at its best and may leave the impression that it is an automatic theorem prover.

Chapter 9 is called **How to Use the Theorem Prover** but might also be titled **How To Use the Theorem Prover to Discover Proofs** or **How to Interact with the Theorem Prover** or **How to Read Theorem Prover Output**. The chapter discusses how experienced users interact with the system. We start by revisiting the demonstration theorems in Chapter 8. However, this time, instead of letting the theorem prover show off, we use it as we would to explore new ground. Producing a proof in this style is really a more interactive, cooperative endeavor in which the theorem prover helps the user find a proof. Or is it the other way around? The chapter also describes how the experienced user tends to use a text editor as a key component in his or her hierarchical development of a complex proof. Despite its brevity and lack of precision—much of its advice is given by example only—this chapter may be the most valuable one in the handbook because it encourages you to think about the theorem prover in a different way than you might otherwise.

Chapter 10 is a description of how the theorem prover works, with particular attention on those aspects of its behavior under the control of the user through previously proved theorems. We also discuss the use of clausal form in the internal representation of formulas and the notion of the “type” of a term in the logic. This chapter is an important prerequisite for the next one.

Chapter 11 describes the various kinds of rules used by the system, how each kind of rule affects the behavior of the system, and how rules are generated from formulas. Understanding this material is a key to the effective use of the theorem prover.

Chapter 12 is the reference guide to the theorem prover. It is organized as a long series of sections, each section titled by and devoted to a single command

or concept. The sections are listed alphabetically. You should probably read the reference guide once straight through. Thereafter, you will probably access it randomly via the table of contents or the index of this handbook.

Chapter 13 is just a long collection of hints on how to use the system. We explain the kinds of definitions the system “likes” and those that it is “biased against.” We give some guidelines on the effective use of the various kinds of rules. In general, the chapter is a random collection of vague, airy-fairy, and contradictory remarks. If we could make it precise and accurate we would simply mechanize all this advice and omit the chapter. As things stand, we leave to you the problem of sorting it all out.

Chapter 14 is a description of how to create an executable version of the theorem prover at your local site from the sources we provide.

The handbook has three appendices.

The first, Appendix I, illustrates the use of the logic as a programming language by defining a function that parses strings of ASCII characters into the formulas of the logic. The appendix serves the additional purpose of formalizing the syntactic conventions of the logic.

The second, Appendix II, lists the axioms for our primitive data types, namely the natural numbers, ordered pairs, literal atoms, and the negatives. These axioms are generated from the shell axiom schemas given in Chapter 4 and are included for two reasons. First, since they illustrate the axiom schemas they might help readers understand the schemas better. Second, the reader who wishes to ignore the provision for user-defined types could substitute these axioms for the “shell principle” and simplify the formal logic.

The third, Appendix III, gives some statistics on the relative complexities of the theorems and proofs explored with the theorem prover during its first 20 years.

2 A Primer for the Logic

Formally, a logic consists of three parts: (a) a language in which formulas are written, (b) a set of distinguished formulas called *axioms*, and (c) a set of transformations, called *rules of inference*, with which one can derive new formulas from old. A *theorem* is any formula that is either an axiom or can be derived from theorems using the rules of inference. Our logic also includes several *extension principles* with which one can soundly extend the logic to include axioms about new function symbols.

The notions mentioned above are entirely *syntactic*. It is possible to discuss the logic from a *semantic* point of view. This involves attaching a *function* to each *function symbol*. Then, given an assignment of values to the *variable symbols* of a term or formula, it is possible to determine the *value* of the expression by evaluation. Given an interpretation of the function symbols, a formula is said to be *valid* if its value is “true” under all possible assignments of values to its variable symbols. A *model* of a logic is an interpretation under which all of the axioms are valid. Note that if all axioms are valid and the rules of inference are validity preserving, then every theorem is valid. Informally, if a formula is a theorem then it is “always true.” There is a model for the logic presented here.

The logic presented here contains three “extension” principles under which the user can add new axioms to the system with the guarantee that the model for logic can be extended to accommodate the new axioms. The principles permit the axiomatization of “new” data types, the definition of “new” recursive functions, and the introduction of constrained functions.

Because of these extension principles it is not meaningful to speak of *the* logic in the traditional sense that implies the choice of some fixed set of axioms.

We more often speak of the *primitive* or *Ground Zero* logic and of various extensions obtained by adding new data types and definitions.

Almost all of the functions axiomatized in the logic are *total recursive*. That is, given an application of a function to explicit values, we can determine algorithmically the explicit value which is provably equal, under the axioms, to the given application. The exceptions to the totality claim all have to do with the provision in the logic of an interpreter for the partial recursive functions.

The mechanization of the logic includes commands for evaluating functions on explicit values. Thus, the logic and its mechanization provide an *applicative programming language* and *execution environment*. On applications of partial functions, the execution environment may “run forever” (until one of the finite system resources is exhausted), but if an answer is returned it will be correct under the axioms. Furthermore, using the theorem prover it is often possible to *prove* that such an application “runs forever.”

In this chapter we informally present the Ground Zero logic as a programming language. We describe the formal language in which formulas are written and give the most common of our abbreviation conventions. We describe the “input/output behavior” of each of the functions in the Ground Zero logic, and we describe the three extension principles. The formally inclined reader can think of this chapter as an informal description of the semantics of the logic and a semantic justification of the extension principles.

In Chapter 4 we give a complete and precise description of the language, the axioms, the rules of inference, and the extension principles. That chapter is completely self-contained. We believe that the experienced user of the system will find questions about the logic answered far more quickly and clearly in that chapter than in this primer. However, we believe the new user will find the primer a better introduction to the logic.

2.1. Syntax

Statements in the logic are called “terms.” A term is either a variable symbol or else is the application of a function symbol of n arguments to n terms.

The variable and function symbols are strings of uppercase alphanumeric characters and certain signs, namely

$$\text{\$ } \wedge \ \& \ * \ _ \ - \ + \ = \ \sim \ \{ \} \ ? \ < \ >$$

Variable and function symbols must start with an alphabetic character. Thus, **x**, **PLUS**, **ADD1**, and **PRIME-FACTORS** are symbols, and **Prime-factors**, **A[x]** and **1E3** are not.

We use the prefix syntax of Church to write down terms. For example, we write **(PLUS I J)** where others might write **PLUS(I,J)** or **I+J**. We permit

“white space”—spaces, tabs, and carriage returns—to occur arbitrarily between symbols and parentheses.

Here is a term, displayed in three different arrangements.

```
(EQUAL N (PLUS (REMAINDER N K)
                (TIMES K (QUOTIENT N K))))
```

```
(EQUAL N
  (PLUS (REMAINDER N K)
    (TIMES K (QUOTIENT N K))))
```

```
( EQUAL N ( PLUS ( REMAINDER N K
) ( TIMES K ( QUOTIENT N K ) ) ) )
```

In the definition of a term, we required that functions be applied to the correct number of argument terms. The claim that the expression above is a term implicitly informs the reader that **EQUAL**, **PLUS**, **REMAINDER**, **TIMES**, and **QUOTIENT** are all function symbols of two arguments. The following expressions are not terms:

```
(EQUAL X)
```

```
(EQUAL X Y Z)
```

Technically, **(PLUS X Y Z)** is not a term because **PLUS** takes two arguments.

Comments can be written in a term, by preceding them with a semicolon. All text from the semicolon up to but not including the next end-of-line is ignored. Thus, here is a term:

```
(IF (ZEROP N)          ;If N is 0
  N                    ;return N
  (ADD1 N))           ;otherwise, return N+1
```

As new functions and objects are introduced into the logic, we introduce abbreviation conventions to permit their succinct expression.

2.2. Boolean Operators

(TRUE)	A constant we abbreviate with T . T is used as the “true” truth value in the functional analogues of the Boolean operators.
(FALSE)	A constant we abbreviate with F . F is the “false” truth value in the functional analogues of the Boolean operators. T and F are distinct.
(IF X Y Z)	If X is F , return Z ; otherwise return Y . We call the first argument of an IF -expression the <i>test</i> , and we call the second and third arguments the <i>true</i> and <i>false branches</i> of the IF , respectively. Note well that the test is compared to F and if different from F the true branch is taken. Thus, since (as we shall see later) 0 is different from F , (IF 0 Y Z) is Y .
(EQUAL X Y)	Return T if X=Y and F otherwise.
(TRUEP X)	Return T if X=T and F otherwise.
(FALSEP X)	Return T if X=F and F otherwise.
(AND P Q)	Return T if both P and Q are non- F ; otherwise return F . Note: the “truth table” for AND is:

Because **AND** is defined with **IF**, any non-**F** input produces the same output as **T**. Thus, **(AND 3 0) = T**.

(OR P Q)	Return T if either P or Q is non- F ; otherwise return F .
(NOT P)	Return F if P is non- F ; otherwise return T .
(IMPLIES P Q)	Return T if either P is F or Q is non- F ; otherwise return F .

2.3. Data Types

One of the ways the user of the logic can extend it is by the addition of axioms to define a “new” class of inductively constructed objects. The *shell principle* provides this facility. From the perspective of the logic as a programming language, the shell principle is a mechanism by which the user can create new

data types. But the principle is also used in the initial creation of the Ground Zero logic to axiomatize the natural numbers, ordered pairs, literal atoms, and the negatives. We discuss these “primitive” data types as examples of the use of the shell principle after we have described the general principle.

The principle permits the axiomatization of typed n -tuples with type restrictions on each of the n components. The axioms necessary to characterize the new type are generated from axiom schemas by instantiating them with names provided by the user.

Below we show the general form of an invocation of the shell principle. The lower case words in typewriter font used below, namely, **const**, **n**, **base**, **r**, **ac_i**, **tr_i**, and **dv_i**, would, in an actual invocation, be filled in by the user with particular function names, numbers, etc.

Shell Definition.

Add the shell **const** of n arguments
with (optionally, base function **base**)
recognizer function **r**,
accessor functions **ac₁**, ..., **ac_n**,
type restrictions **tr₁**, ..., **tr_n**, and
default functions **dv₁**, ..., **dv_n**.

The above incantation adds axioms to the logic to define a new “data type.” Objects of the new type are “recognized” by the newly axiomatized function **r**, which returns **T** or **F** according to whether its argument is of the new type. The “base function,” **base**, takes no arguments and returns an object of the new type. The “constructor function,” **const**, takes n arguments and returns an n -tuple of the new type. The n functions **ac₁**, ..., **ac_n** “access” the respective components of constructed n -tuples of the new type. The n “type restrictions,” **tr₁**, ..., **tr_n**, describe what types of objects are in the components of each constructed n -tuple. The n “default functions,” **dv₁**, ..., **dv_n**, are constant functions that specify the values to be used when the supplied argument fails to satisfy its type restriction. Every object of the new type is either the base object, (**base**), or else is constructed by **const** from smaller objects satisfying the type restrictions.

Here is a sample shell definition:

Shell Definition.

Add the shell **ADD1** of one argument
with base function **ZERO**,
recognizer function **NUMBERP**,
accessor function **SUB1**,
type restriction (**ONE-OF NUMBERP**),
default function **ZERO**.

By this invocation of the shell principle the function **NUMBERP** is axiomatized to recognize “new” objects constructed from the base object (**ZERO**) by the function **ADD1**. Given a non-(**ZERO**) **NUMBERP**, (**ADD1 n**), the function **SUB1** returns the single constituent, **n**. **ADD1** “expects” its argument to be a **NUMBERP** and coerces it to (**ZERO**) if it is not. The **NUMBERPs** are our formalization of the natural numbers constructed from 0 ((**ZERO**)) by the “successor function” **ADD1**. **SUB1** is the “predecessor function.”

There are restrictions on how the shell principle can be invoked. These restrictions ensure that the axioms consistently extend the logic and describe a nonempty set of “new” objects.

When the shell principle is invoked, **const**, **base**, **r** and each of the **ac_i** must be new function symbols, i.e., never before used. The shell principle effectively defines these symbols.

The default functions must either be **TRUE**, **FALSE**, previously introduced base function symbols, or the new base function.

The language for describing the “type restriction” on the components is very primitive. Each type restriction requires of its respective component either that it be of a type among an explicitly given set of types or that it be of a type not among an explicitly given set of types. In particular, each type restriction is a sequence of the form (**flg r₁ ... r_m**) where **flg** is either **ONE-OF** or **NONE-OF** and each **r_i** is either a previously introduced recognizer, **TRUEP**, **FALSEP**, or the new recognizer.

For example, if **NUMBERP** and **LISTP** are among the “current types,” then the type restriction (**ONE-OF NUMBERP LISTP**) is satisfied only by objects of type **NUMBERP** or **LISTP**, while the type restriction (**NONE-OF NUMBERP**) is satisfied by objects of any type but **NUMBERP**. Note that (**NONE-OF**) is satisfied by objects of any type. If no type restrictions are provided in the invocation, we use (**NONE-OF**).

After the shell principle has been properly invoked, **const**, **base**, **r**, and each of the **ac_i** may be described as follows—following our already established convention for documenting the “behavior” of functions in the logic:

(**const x₁ ... x_n**)

If **x_i** satisfies type restriction **tr_i**, for each $1 \leq i \leq n$, then return an **n**-tuple of the new type, **r**, with the **x_i** as the respective components. If any **x_i** fails to satisfy its type restriction, the corresponding component in the constructed **n**-tuple is (**dv_i**). We call **const** the *constructor* function of the new type.

(**base**)

Return the “base object” of the new type, **r**. That is, (**base**) is an object of the new type but not one constructed by the constructor. If no base object is provided in

the invocation, every object of type \mathbf{x} is constructed by the constructor.

($\mathbf{x} \ \mathbf{x}$) If \mathbf{x} is of the new type, \mathbf{x} , return \mathbf{T} ; otherwise return \mathbf{F} .
($\mathbf{ac}_i \ \mathbf{x}$) If \mathbf{x} is an n -tuple of the new type, \mathbf{x} , constructed by the constructor, **const**, return its i th component; otherwise, return **(\mathbf{dv}_i)**.

Each shell class is disjoint from all others. In addition, \mathbf{T} and \mathbf{F} are different from all shell objects.

It is convenient, in fact, to think of \mathbf{T} and \mathbf{F} merely as the base objects of two shell classes recognized by **TRUEP** and **FALSEP** respectively. These two shell classes have no constructors. The user of the logic cannot use the shell principle to axiomatize a class with no constructor.

The Ground Zero theory includes four shells axiomatized with the shell principle:

- **NUMBERP** - the natural numbers
- **LISTP** - the ordered pairs
- **LITATOM** - the words
- **NEGATIVEP** - the negative integers

In the following four subsections we show the invocation of the shell principle used for each of these types and describe the functions axiomatized.

2.3.1. *Natural Numbers*

With the invocation

Shell Definition.

Add the shell **ADD1** of one argument
 with base function **ZERO**,
 recognizer function **NUMBERP**,
 accessor function **SUB1**,
 type restriction **(ONE-OF NUMBERP)**, and
 default function **ZERO**.

we define the following functions:

(ZERO) A constant, abbreviated **0**. This is the “base object” of the natural number shell.

- (**ADD1** *I*) Return the next natural number after *I*. This is the “constructor” of the natural number shell. If *I* is not a natural, 0 is used in its place.
- (**NUMBERP** *X*) Return **T** or **F** according to whether *X* is a natural number. This is the “recognizer” of the natural number shell.
- (**SUB1** *I*) Return the predecessor of the natural number *I*. If *I* is 0 or is not a natural number, return 0. This is the “accessor” of the natural number shell.

2.3.2. *Ordered Pairs*

With the invocation

Shell Definition.

Add the shell **CONS** of two arguments with recognizer function **LISTP**, accessor functions **CAR** and **CDR**, and default functions **ZERO** and **ZERO**.

we define the following functions:

- (**CONS** *X Y*) Return the ordered pair whose first component is *X* and whose second is *Y*.
- (**LISTP** *X*) Return **T** or **F** according to whether *X* is an ordered pair constructed by **CONS**.
- (**CAR** *X*) If *X* is an ordered pair, return the first component; otherwise, return 0.
- (**CDR** *X*) If *X* is an ordered pair, return the second component; otherwise, return 0.

Nests of ordered pairs are used to represent finite sequences, binary trees, tables, etc. We will later explain abbreviation conventions for nests of ordered pairs. When we do that we will illustrate the use of such pairs.

2.3.3. *Literal Atoms*

Literal atoms are used to represent symbolic data such as words. We will later introduce an abbreviation for writing literal atoms that makes their use obvious. For the moment, we will simply describe the primitives for constructing, recognizing, and accessing literal atoms.

With the invocation

Shell Definition.

Add the shell **PACK** of one argument
with recognizer function **LITATOM**,
accessor function **UNPACK**,
default function **ZERO**.

we define the following functions:

- (**PACK L**) Construct and return the literal atom with **L** as its “print name.” Generally, **L** is a list of ASCII character codes, but it may be any object.
- (**LITATOM X**) Return **T** or **F** according to whether **X** is a literal atom.
- (**UNPACK X**) If **X** is a literal atom obtained by **PACK**ing **L**, return **L**; otherwise return **0**.

After explaining the extended syntax, we illustrate how **LITATOMs** are used to represent words.

2.3.4. *Negative Integers*

With the invocation

Shell Definition.

Add the shell **MINUS** of one argument
with recognizer function **NEGATIVEP**,
accessor function **NEGATIVE-GUTS**,
type restriction (**ONE-OF NUMBERP**), and
default function **ZERO**.

we define the following functions:

- (**MINUS I**) If **I** is a natural number, return the negative of **I**.
- (**NEGATIVEP X**) Return **T** or **F** according to whether **X** is a negative integer.
- (**NEGATIVE-GUTS X**) If **X** is a negative integer, return the absolute value of **X**; otherwise, return **0**.

The function **MINUS**, when given **0**, creates an unusual “negative” number whose “absolute value” is **0**. That is to say, (**NEGATIVEP (MINUS 0)**) = **T**. Thus, (**EQUAL (MINUS 0) 0**) = **F**.

2.3.5. *Some Intuitive Remarks about Shells*

A very concrete model for shells can be described as follows. Each shell class has a color assigned to it: **true** is black, say, **false** is white, natural numbers are red, lists are blue, literal atoms are yellow, and negatives are green. The recognizers are filters that accept only a given color: **TRUEP** returns **T** when presented a black object and **F** otherwise; **FALSEP** returns **T** when presented a white object and **F** otherwise; **NUMBERP** returns **T** when given a red object and **F** otherwise, etc. The base objects of each class are marbles of the appropriate color: **T** is a black marble, **F** is a white one, and **0** is a red one. The constructor functions build colored, compartmentalized boxes, with as many compartments as the constructors take arguments: **ADD1** builds red boxes with one compartment, **CONS** builds blue boxes with two compartments, **PACK** builds yellow boxes with one compartment, and **MINUS** builds green boxes with one compartment. Roughly speaking, a constructor function puts each of its arguments into the corresponding compartments of the constructed box. But first it asks whether the presented argument is of the right color: **ADD1** expects its argument to be red, **CONS** and **PACK** don't care, and **MINUS** expects its argument to be red. When a constructor is given the wrong color argument it puts the corresponding default object into the compartment instead. (Each default object is a marble of the expected color.) Finally, the accessors are functions that expect to be given boxes of the appropriate color and, when they are, they merely extract the contents of the corresponding compartment. When an accessor is given an object that is either not a box or is of the wrong color, it returns the corresponding default value.

Thus, the number **2** in this model, **(ADD1 (ADD1 (ZERO)))**, is a red box containing a red box containing a red marble. The ordered pair **(CONS 2 (CONS 0 0))** is a blue box containing two compartments; in the first is a 2 (i.e., a red box containing...); in the second is a blue box containing two compartments; in each is a red marble. What is **(ADD1 T)**? According to the model, **ADD1** builds a red box containing a red... oops! The argument provided, **T**, is black. **ADD1** expects a red argument. So it uses a red marble instead. Thus, **(ADD1 T)** is a red box containing a red marble—the same thing constructed by **(ADD1 0)** or **1**. What is **(SUB1 (ADD1 T))**? Again, **(ADD1 T)** constructs a box containing a red marble. **SUB1** extracts the contents of the compartment, the red marble, and returns it. **(SUB1 (ADD1 T))** is a red marble, i.e., **0**, not **T**.

These intuitive remarks can be formalized to provide a model for the shell axioms, but we do not go into that in this handbook. The name “shell” came from the idea of constructor functions building colored containers, or shells, around their arguments.

Any extension of the theory will contain only a finite number of explicitly declared shells. In Ground Zero we have six shells, counting **T** and **F** among them. One might therefore ask “is every object in the model of one of the given shells?” Put more informally, “are the only colors black, white, red, blue, yellow, and green?” The answer is “no.” If the answer were “yes, there are only six colors,” then the theory would be inconsistent as soon as we invoked the shell principle again to axiomatize purple triples.

Finally, new users of the logic are often surprised and baffled to see the natural numbers axiomatized in a way analogous to ordered pairs. But **SUB1** is to **ADD1** exactly what **CAR** is to **CONS**. Table 2.1 illustrates the correspondences. That is, **ADD1** constructs a 1-tuple recognized by **NUMBERP** and ac-

Table 2.1

const	(base)	r	accessors
-	(TRUE)	TRUEP	-
-	(FALSE)	FALSEP	-
ADD1	(ZERO)	NUMBERP	SUB1
CONS	-	LISTP	CAR, CDR
PACK	-	LITATOM	UNPACK
MINUS	-	NEGATIVEP	NEGATIVE-GUTS

cessed by **SUB1**. **CONS** constructs a 2-tuple, recognized by **LISTP** and accessed by **CAR** and **CDR**. **28** is a red box containing **27**. **(CONS T 0)** is a blue box containing **T** and **0**. **27** is the first component of **28** in exactly the same sense that **T** is the first component of **(CONS T 0)**. However, red boxes have only one compartment while blue ones have two.

We now return to the mainstream of our presentation of the logic by continuing the documentation of the primitive functions.

2.3.6. COUNT

The function **COUNT** determines the “size” of a shell object. In particular, given an object (actually, an explicit value) constructed entirely by applications of shell constructor and base functions, **COUNT** returns the number of constructor function applications.

(COUNT X) If **X** is **T** or **F**, return **0**; if **X** is the base object of some shell, return **0**; if **X** is a constructed n-tuple, return **1** plus the sum of the **COUNT**s of the n components.

For example, let **x** be the constant

```
(CONS (ADD1 (ADD1 (ZERO)))
      (CONS (PACK (CONS (ZERO) (ZERO)))
            (ZERO)))
```

Then, (**COUNT x**) is **6**. Put in the metaphorical terms of the previous subsection, when **COUNT** is presented with an object composed entirely of boxes and marbles, **COUNT** returns the number of boxes.

COUNT is unusual in that its definition “grows” as new shells are defined. The user of the theory cannot define a function like this. The reason is that in order to permit “data abstraction,” the only way to obtain the *n* components of a shell object is to apply the *n* accessor functions. Thus, the user who tries to define a function like **COUNT** will find himself enumerating each of the known shells. Such a definition would be incomplete when a new shell is added.

2.4. Extending the Syntax

We adopt several conventions to permit the succinct expression of shell constants.

2.4.1. Elementary Abbreviations

0 is an abbreviation of (**ZERO**); the positive integer **n** is an abbreviation of the nest of **n** **ADD1**s around a **0**; the negative integer **-n** is an abbreviation of (**MINUS n**). Thus **2** abbreviates (**ADD1 (ADD1 (ZERO))**) and **-4** abbreviates (**MINUS 4**).

We permit integers to be written in four different bases: decimal (10), binary (2), octal (8), and hexadecimal (16). No special syntax is required for decimal notation: merely write the string of decimal digits, optionally preceded by a plus or minus sign and optionally followed by a decimal point. Thus, **123**, **123.**, **+123**, and **+123.** all denote the same integer and, when used as terms, abbreviate a nest of one hundred and twenty-three **ADD1**s around a (**ZERO**).

To write an integer in binary notation, write a number sign (**#**) followed by the character **B**, followed by the optionally signed string of **1**s and **0**s expressing the integer in base 2. Thus, **#B100** is the integer 4 in binary notation; **#B-100** is the integer -4. When used as a term, **#B-100** abbreviates the same term as (**MINUS 4**).

Octal notation is similar to binary except that the letter **O** (as in “octal”) follows the number sign and the digits allowed are **0** through **7**. Thus, **#O100** is a way of writing the integer 64.

Finally, hexadecimal notation uses the letter **X** after the number sign and allows the sixteen digits (listed here in ascending order) **0**, **1**, ..., **9**, **A**, **B**, **C**, **D**, **E**, and **F**. Thus, **#X-FAB** is a way of writing the integer -4011 ($= -(15 \cdot 16^2 + 10 \cdot 16^1 + 11 \cdot 16^0)$).

We permit certain function symbols to be applied to “too many” arguments by right associating the function. The function symbols in question are **AND**, **OR**, **PLUS**, and **TIMES**. For example **(AND P Q R)** is an abbreviation for **(AND P (AND Q R))**.

Nests of **CARS** and **CDRs** are abbreviated with function symbols of the form **C...A...D...R**, e.g., **(CADAAR X)** is an abbreviation of **(CAR (CDR (CAR (CAR X))))**.

Because **CONS** nests are used to represent many structures, it is convenient to let **(LIST* x₁ x₂ ... x_n x_{n+1})** be an abbreviation for **(CONS x₁ (CONS x₂ ... (CONS x_n x_{n+1})...))**. Thus, for example, **(LIST* 1 2 0)** is an abbreviation for **(CONS 1 (CONS 2 0))**.

IF-expressions are also commonly nested and so we use **(COND (t₁ v₁) (t₂ v₂) ... (t_n v_n) (T v_{n+1}))** as an abbreviation for **(IF t₁ v₁ (IF t₂ v₂ ... (IF t_n v_n v_{n+1})...))**. Each “argument” to **COND** is called a “**COND** clause” and must contain exactly two expressions, a test and a value. The final test must be **T**.

When each test of a conditional expression is an equality in which a fixed “key” is compared to a constant—a common form of case analysis—the following abbreviation is useful.

```
(CASE key (e1 v1)
          (e2 v2)
          ...
          (en vn)
          (OTHERWISE vn+1))
```

is an abbreviation for

```
(COND ((EQUAL key (QUOTE e1)) v1)
      ((EQUAL key (QUOTE e2)) v2)
      ...
      ((EQUAL key (QUOTE en)) vn)
      (T vn+1))
```

We will explain what the **QUOTE** notation means shortly. For the moment it is sufficient to know that, if well formed, **(QUOTE e)** abbreviates a constant. Each “argument” to **CASE**, except for the first, is called a “**CASE** clause” and must consist of an “explicit value descriptor” (something that Nqthm permits you to write “inside a **QUOTE**”) and a value. All of the descriptors must be distinct and the final descriptor must be **OTHERWISE**. Intuitively, a **CASE** expression is equal to v_i if the value of **key** is the constant denoted by e_i and is v_{n+1} otherwise.²

(LET ((v₁ t₁) ... (v_n t_n)) body), where the v_i s are distinct variable symbols and the t_i s and **body** are terms, is an abbreviation for the term obtained by simultaneously replacing in **body** all “free” occurrences of each v_i by the corresponding t_i . Thus, **(LET ((X (SORT Y))) (EQUAL (SORT X) X))** is an abbreviation for **(EQUAL (SORT (SORT Y)) (SORT Y))**. Observe that while **LET** appears to introduce “bound” variables, e.g., **X** in the preceding example, it actually does not because the **LET** expression is just an abbreviation. If abbreviations are eliminated in **body** before the substitution is done, all variables occurring in the “translated” **body** are “free.”

2.4.2. QUOTE Notation

We next describe the **QUOTE** notation, with which list and atomic constants are written down. **QUOTE** notation is often confusing to newcomers, but a logic in which you cannot succinctly express large structures is a mere toy.

In this notation, the symbol **QUOTE** is used in a way that makes it appear as though it were a function of one argument; typical expressions include **(QUOTE ABC)** and **(QUOTE (1 2 3))**. However, the “argument” expression is not necessarily a term. While the word “**QUOTE**” is a legal symbol and hence, technically, a legal function symbol, we *never* use **QUOTE** as a function symbol. The entire **QUOTE** expression—the string of characters in the “argument” position, the word **QUOTE**, and the enclosing parentheses—is an abbreviation for a term.

We now ask the reader to adopt an abbreviation convention that has nothing whatsoever to do with terms. Henceforth, whenever you see the single quote mark, (**'**), immediately preceding a symbol or well-balanced parenthesized string, **s**, pretend you saw **(QUOTE s)** in place of the single quote mark and **s**. Thus, **'ABC** should be read as **(QUOTE ABC)**, **'(1 B 3)** should be read as

²Readers familiar with Common Lisp may be troubled by the fact that Nqthm treats **(CASE X ((A B C) T) (OTHERWISE F))** as **(IF (EQUAL X '(A B C)) T F)** rather than as **(IF (MEMBER X '(A B C)) T F)**.

(QUOTE (1 B 3)), and '(A 'B ''C) should be read as

(QUOTE (A (QUOTE B) (QUOTE (QUOTE C)))).

We now proceed to explain what these strings mean.

If **word** is a sequence of ASCII characters satisfying the syntactic rules for a symbol in our logic and the ASCII codes for the successive characters in **word** are c_1, \dots, c_n , then (QUOTE **word**), i.e., '**word**', is an abbreviation for

(PACK (LIST* $c_1 \dots c_n$ 0)).

Thus, 'ABC is an abbreviation of (PACK (CONS 65 (CONS 66 (CONS 67 0)))).

'NIL, which is an abbreviation for

(PACK (CONS 78 (CONS 73 (CONS 76 0)))),

is further abbreviated **NIL**. Recall that the symbol **NIL** is not a variable symbol. Note that the object **NIL** is a **LITATOM** and, hence, is not **F**. Thus, to the surprise of many Lisp programmers, (IF **NIL** **Y** **Z**) = **Y**.

Because **NIL** is so often used to terminate lists, we adopt the abbreviation that (LIST $x_1 x_2 \dots x_n$) stands for (LIST* $x_1 x_2 \dots x_n$ **NIL**). (LIST) is an abbreviation for **NIL**.

Table 2.2

x	abbreviation of x
(CONS 1 (CONS 2 NIL))	'(1 2)
(CONS 1 (CONS 2 'ABC))	'(1 2 . ABC)
(CONS (CONS 'A 1) (CONS (CONS 'B 2) (CONS (CONS 'C 3) NIL)))	'((A . 1) (B . 2) (C . 3))
(CONS (CONS 'A 'B) (CONS 'C 'D))	'((A . B) C . D)
(CONS 'EQUAL (CONS 'X (CONS ''ABC NIL)))	'(EQUAL X (QUOTE ABC))

Finally, we provide a convention for abbreviating certain **LISTP** constants. The convention is similar to the “dot” notation of Lisp. The notation is extremely succinct but frequently causes trouble for newcomers. Table 2.2 illustrates our conventions. We so abbreviate any term built entirely with **CONSES** from numbers (i.e., terms built with **ADD1s**, **ZEROs**, and **MINUSES**) and those **LITATOMs** (i.e., terms built with **PACKs**) admitting the abbreviation convention noted above. If **x** is such a term, we abbreviate **x** by a single quote mark (') followed by the “explicit value descriptor” of **x** as defined below. If **x** is a **NUMBERP** or **NEGATIVEP** abbreviated by **n**, its explicit value descriptor is **n**. If **x** is a **LITATOM** abbreviated by **'wrd**, its explicit value descriptor is **wrd**. Otherwise, **x** is a **LISTP** and of the form **(CONS x₁ (CONS x₂ ... (CONS x_n fin)...))** where **fin** is the first non-**CONS** expression in the **CDR** chain of **x**. If **fin** is **NIL**, then the explicit value descriptor of **x** is an open parenthesis followed by the explicit value descriptor of **x₁**, a space (or arbitrary amount of white space), the explicit value descriptor of **x₂**, a space, ... the explicit value descriptor of **x_n**, and a close parenthesis. If **fin** is non-**NIL**, then the explicit value descriptor of **x** is as it would be had **fin** been **NIL** except that immediately before the close parenthesis there should be inserted some space, a dot, some space, and the explicit value descriptor of **fin**.

The reader unfamiliar with dot notation is urged to inspect Table 2.2 in light of the above definition.

Actually, our notation is somewhat more elaborate than indicated above because it is possible to include **T**, **F**, and new shell constants inside quoted objects. However, we do not give the details here.

The last example in Table 2.2 hints at one of the most important uses of quoted constants in the logic: the representation of terms in the logic. By so embedding the terms of the language into the logic we can define an interpreter for the logic as a function in the logic and provide many features that otherwise would be considered “higher order.”

2.4.3. Backquote Notation

Consider the list

```
'((A . 1) (B . 2)).
```

Suppose we wished to write down a similar list, but one that has the values of the variables **X** and **Y** where the list above has **1** and **2**, respectively. This list cannot be written in **QUOTE** notation because it is not a constant—the list we have in mind is a function of **X** and **Y**. If we knew that the value of **X** was some constant **'x** and the value of **Y** was **'y**, then we could write the new list:

`'((A . x) (B . y)).`

But in the absence of the knowledge of the values of **X** and **Y** we must use **LIST** and **CONS** instead:

`(LIST (CONS 'A X) (CONS 'B Y)).`

After getting used to **QUOTE** notation, the use of **LIST** and **CONS** to describe such “near constants” is awkward, especially for large structures.

In Lisp this problem is solved by the introduction of “backquote notation.” See, for example, page 527 of [132]. We adopt a version of backquote notation. The notation allows one to write constants exactly as in **QUOTE** notation but also provides an “escape” mechanism that permits parts of the “constant” to be the values of arbitrary terms. In particular, it lets us write the list we had in mind above as as

``((A . ,X) (B . ,Y)).`

That is, we write a backquote instead of the single quote mark and then exhibit the structure of the “constant” in question, with the added freedom that if we wish to express some non-constant component we can escape from the constant notation by writing a comma followed by an arbitrary term.

Of course, the term abbreviated by the backquote expression above is not a constant, even though it sort of looks like one. It is a term involving the variables **X** and **Y** and certain list construction functions. The backquote notation is just a convenient way to write certain list expressions.

As illustrated above, the comma provides one escape from the constant notation and means “here put the value of the following term.” A second escape is provided. It can only be used in positions where a list element is normally written and means “here splice in as elements all the elements in the list produced by the following term.” The syntax for this escape is a comma followed by an @-sign. Thus, ``(A B ,@L C D)` is an abbreviation for `(CONS 'A (CONS 'B (APPEND L (CONS 'C (CONS 'D 'NIL)))))`. **APPEND**, defined later, concatenates its two arguments.

Here is a more complicated example of backquote notation. The expression

``(IMPLIES (AND ,@HYPS) (EQUAL ,LHS ,RHS))`

is an abbreviation for the same term denoted by

`(LIST 'IMPLIES
 (CONS 'AND (APPEND HYPS NIL))
 (LIST 'EQUAL LHS RHS)),`

namely

```
(CONS 'IMPLIES
  (CONS (CONS 'AND (APPEND HYPs 'NIL))
    (CONS (CONS 'EQUAL
      (CONS LHS (CONS RHS 'NIL)))
      'NIL)))
```

2.5. Conventional Data Structures

Ordered pairs are used to construct lists or finite sequences, tables, binary trees, and many other data structures. Literal atoms are typically used to represent such symbolic constants as words. In this section we illustrate typical uses of these data structures.

The literal atom **NIL** is conventionally used to represent the empty list. This sometimes causes confusion since **NIL** is not a **LISTP**. Perhaps “**CONSP**” would have been a better name for **LISTP**.

Given the representation of some list **s**, the ordered pair **(CONS x s)** is the conventional representation of the list obtained by adding the new element **x** to the front of **s**. Thus, the list of the first five positive numbers is conventionally represented by

```
(CONS 1 (CONS 2 (CONS 3 (CONS 4 (CONS 5 NIL)))))
```

which may be abbreviated as

```
(LIST 1 2 3 4 5)
```

or

```
'(1 2 3 4 5).
```

If **lst** is a nonempty list, then **(CAR lst)** is the first element in the list, and **(CDR lst)** is the list of the remaining elements. Thus,

```
(CAR '(1 2 3 4 5))      = 1
(CDR '(1 2 3 4 5))      = '(2 3 4 5)
```

```
(CADR '(1 2 3 4 5))
=
(CAR (CDR '(1 2 3 4 5)))
=
(CAR '(2 3 4 5))        = 2
```

```
(CADDR '(1 2 3 4 5))    = 3
(CADDDR '(1 2 3 4 5))   = 4
```

```
(CADDDDR '(1 2 3 4 5))    = 5
(CDDDDDR '(1 2 3 4 5))    = NIL
```

“Head” and “Tail” might be more memorable names for **CAR** and **CDR**, when they are used to access the components of ordered pairs representing lists in the conventional way. It should be stressed that there is nothing inherent in **NIL**, **CAR**, **CDR**, and **CONS** that dictates the choice of **NIL** as the empty list, the choice of **CAR** for the storage of the first element, or **CDR** for the storage of the remaining elements. It is merely a matter of convention. We could as well use the atom **'ABC** to denote the empty list and reverse the roles of **CAR** and **CDR**. In that representation, the list of the first five positives would be written

```
(CONS (CONS (CONS (CONS (CONS 'ABC 5) 4) 3) 2) 1)
```

or

```
'((((ABC . 5) . 4) . 3) . 2) . 1).
```

From this example it should be clear that the **QUOTE** notation *does* encourage the use of the traditional convention for representing lists!

Literal atoms are commonly used to represent symbolic data such as names, operators, and attributes. For example, in an application in which the days of the week are relevant, one might choose to denote each day with a number. But it might be more perspicuous to use literal atoms, e.g., **'MONDAY** for the first work day, **'TUESDAY** for the second, etc.

'MONDAY is, of course,

```
(PACK '(77 79 78 68 65 89 . 0)).
```

It is the **QUOTE** notation, rather than **PACK**, which suggests a connection between **'MONDAY** and the word “monday”.

Here are some examples illustrating how **PACK** and **UNPACK** work, in **QUOTE** notation:

```
(UNPACK 'MONDAY) = '(77 79 78 68 65 89 . 0)
```

```
(PACK (CONS 77 (CDR (UNPACK 'MAN)))) = 'MAN
```

Of course, it is frequently the case that literal atoms are used inside list structures. For example, the work days are

```
'(MONDAY TUESDAY WEDNESDAY THURSDAY FRIDAY).
```


Lists are frequently used to represent tables. A very common form of list is an “association list” which represents a two column table such as that displayed in Table 2.3.

Table 2.3

key	value
ADD1	1
DIFFERENCE	2
PLUS	2
SUB1	1
TIMES	2
ZERO	0

The association list representing Table 2.3 is the list of successive key-value pairs:

```
'((ADD1 . 1)
  (DIFFERENCE . 2)
  (PLUS . 2)
  (SUB1 . 1)
  (TIMES . 2)
  (ZERO . 0))
```

Finally, ordered pairs are commonly used to represent trees. The simplest form of tree is the binary tree. Thus, the tree

might be represented by

```
(CONS (CONS 1 2) (CONS (CONS 3 4) 5))
```

which may be written

`'((1 . 2) (3 . 4) . 5)`

However, with lists one can conveniently represent trees with branching rates other than two. For example, a common representation of the tree shown below, which has “operators” of varying arity stored at the nodes

is the list

`'(PLUS (ADD1 X) (TIMES Y (CRYPT M E N)))`

In this representation, each node of the tree is represented by a pair whose **CAR** holds the information for the node and whose **CDR** is a list of the subtrees. Observe the close similarity between such trees and the terms of our logic. In fact, the **QUOTE** notation makes this similarity strikingly obvious. We call `'(PLUS X Y)` the “quotation” of the term `(PLUS X Y)` and will eventually define a function, called **EVAL\$**, with the property that the **EVAL\$** of `'(PLUS X Y)`, in a suitable environment, is equal to `(PLUS X Y)`.

We will illustrate functions for manipulating these conventional data structures after we have discussed the definitional principle.

2.6. Defining Functions

The user can define new functions by adding equations that permit calls of the new functions to be reduced to calls of old ones. We illustrate the principle of definition many times in this section. However, the functions defined in this section are not part of the Ground Zero theory and are presented for illustrative purposes only.

The equation

Definition .

(STEP X) = (ADD1 (ADD1 X))

defines the new function **STEP**, which takes one argument, **X**, and returns the successor of its successor. Thus,

$$\begin{aligned} (\text{STEP } 5) &= (\text{ADD1 } (\text{ADD1 } 5)) \\ &= (\text{ADD1 } 6) \\ &= 7 \end{aligned}$$

A typical invocation of the definitional principle is

Definition .

(fn x₁ ... x_n) = term

We say that the *formals* of **fn** are **x₁**, ..., **x_n**, and the *body* of **fn** is the term **term**.

When the principle of definition is invoked, **fn**, the formals, and the body must satisfy four restrictions.

1. **fn** must not be already defined or even mentioned in any axiom of the logic.
2. Each of the formals must be a variable, and they must all be distinct.
3. The body must be a term and may contain no variables other than the formals. That is the body may contain no “global” variables.

We shall explain the fourth restriction later. In general the restrictions on definitional equations are present to ensure that there exists one and only one function satisfying the equation. Violating restriction 1 may lead to unsatisfiable constraints on the function “defined.” For example, defining a function named **CAR** will almost certainly lead to inconsistent statements about its properties. One subtle consequence of this restriction is that it prohibits mutually recursive definitions: if **F1** is defined in terms of **F1** and **F2**, then a subsequent attempt to define **F2** will fail because **F2** is already mentioned in an axiom. We show how we get around this prohibition in the next section.

Violating restriction 2, as in the following example

“**Definition.**”

(FN X X) = 2

yields “definitions” that fail to specify the value of the function on all combinations of arguments. For example, what is **(FN 1 2)**?

Violating restriction 3, as in

“**Definition.**”

(FN X) = Y

may yield “definitions” of “functions” that seem to return many different values for the same input. For example, if the equation above were admitted as an axiom then, by instantiation, we could prove **(FN 1) = T**. But we could also prove **(FN 1) = F**. Thus, we could prove **T=F**, which is not true.

Here are some examples of simple function definitions:

Definitions.

(NLISTP X) = (NOT (LISTP X))

(FIRST X) = (CAR X)

(REST X) = (CDR X)

(NULL X) = (EQUAL X NIL)

(SECOND X) = (FIRST (REST X))

(THIRD X) = (FIRST (REST (REST X)))

These definitions illustrate one reason to define a function: to package, under a memorable name, a commonly used nest of function applications. Such definitions do not really add anything to our expressive power since we could always say the same thing by using the old nest of functions. But they make it easier to build complex systems.

The power of function definitions is most apparent when one starts to write *recursive* definitions, i.e., definitions in which the newly defined function symbol is used inside the body of its own definition.

For example, here is an equation that permits us to determine the number of elements in a list:

(LEN L)

=

(IF (NULL L) 0 (ADD1 (LEN (REST L))))

Observe that from this equation we can derive

```

(LEN '(A B C)) = (ADD1 (LEN '(B C)))
                = (ADD1 (ADD1 (LEN '(C))))
                = (ADD1 (ADD1 (ADD1 (LEN NIL))))
                = (ADD1 (ADD1 (ADD1 0)))
                = 3

```

Thus, we actually have introduced a new concept. No nest of “old” functions would yield the same result as **(LEN L)** on every possible **L**. Without an equation such as this we could not talk formally about the “length of a list.”

We generally say a function is *recursive* if it is called inside the body of its own definition; a function is said to be *nonrecursive* otherwise. Thus, in this usage, **LEN** above is recursive and **SECOND** is nonrecursive. This terminology is at odds with the standard mathematical terminology (e.g., in [118]) where “recursive” is equivalent to “computable.” **SECOND**, while nonrecursive in our sense, is certainly computable! Our usage of the terms is entirely consistent with everyday practice in programming—not a very strong recommendation of a convention at odds with mathematics...

The power of recursive definitions comes at a price. Consider the following derivation from the equation “defining” **LEN** above:

```

(LEN 0) = (ADD1 (LEN (REST 0))) ; (NULL 0) = F
        = (ADD1 (LEN (CDR 0)))  ; REST is CDR
        = (ADD1 (LEN 0))         ; (CDR 0) = 0

```

If the equation “defining” **LEN** is merely used as a computational rule, we see that the computation of **(LEN 0)** does not terminate; indeed, **(LEN L)** terminates only if the “last **CDR**” of **L** is **NIL**. But if the equation “defining” **LEN** is available as an *axiom*, we can show that **(LEN 0)** is one greater than itself!

```

(LEN 0) = (ADD1 (LEN 0))

```

But from the definition of **ADD1** and the natural numbers we can show that for all **X**, **X** ≠ **(ADD1 X)**! Hence, by instantiating **X** to be **(LEN 0)** we can get

```

(LEN 0) ≠ (ADD1 (LEN 0)).

```

We can thus prove that **(LEN 0)** both is and is not **(ADD1 (LEN 0))**. This is an intolerable state of affairs for a logic: adding a “definition” rendered the logic inconsistent. The problem arose because there is no function satisfying the alleged definition.

The fourth restriction on definitional equations prevents this problem. Ideally we would like an easily enforced syntactic requirement that admitted every equation that defines a function and prohibited any equation that does not. Unfortunately, no such requirement can exist. Our fourth restriction is, however, sufficient and does not often prohibit useful definitions: the recursion

in the definition must be proved to terminate. It can be shown that if the recursion terminates, then there exists a function satisfying the equation—namely, the function computed by an ideal computing machine which follows the recipe laid out in the definition.

To formalize restriction 4, we formalize the notion of the ordinals up to ε_0 and the well-founded relation **ORD-LESSP** on those ordinals. We then require

4. There must exist an ordinal measure of the arguments of **fn** that can be proved to decrease according to **ORD-LESSP** in each recursive call of **fn**.

We discuss the ordinals later. However, the natural numbers constitute the initial ordinals and, on two natural numbers, **ORD-LESSP** is equivalent to the familiar “less than” relation.

Violating restriction 4, as in the “definition” of **LEN** above, often (but not always) produces inconsistent axiomatizations. Here is an acceptable definition of a similar concept:

Definition.

```
(LENGTH L)
=
(IF (LISTP L)
    (ADD1 (LENGTH (CDR L)))
    0).
```

A measure satisfying restriction 4 in this case is **(COUNT L)**. The recursive branch is taken only if **L** is a **LISTP** and it can be shown that if **(LISTP L)** then **(COUNT (CDR L))** is strictly smaller than **(COUNT L)**. More formally, we have

Theorems.

```
(ORDINALP (COUNT L))
```

and

```
(IMPLIES (LISTP L)
          (ORD-LESSP (COUNT (CDR L))
                     (COUNT L)))
```

Note that **(LENGTH 0) = 0** since **0** is not a **LISTP**.

It is very common for a single argument of a recursive function to get smaller in each recursion; **COUNT** is the most commonly used measure function. Note carefully, however, that the measure justifying a definition is an arbitrary term which can take into account all of the arguments, not just a single one. While some measure of the arguments must decrease, it may be that no single argument gets smaller.

Below we illustrate the point just made. Consider the function that returns a list of the natural numbers from **I** to **J**. We call the function **FROM-TO**

Definition.

```
(FROM-TO I J)
=
(IF (LESSP J I)
    NIL
    (CONS I (FROM-TO (ADD1 I) J))).
```

If **J** is **LESSP** (“less than”) **I**, the answer is the empty list. Otherwise, the answer is obtained by “consing” **I** onto the naturals from **I+1** to **J**. That is, we collect the naturals from **I+1** to **J** and then add **I** to the front. Observe that one argument increases in each recursion while the other stays constant. What is getting smaller?

This definition is acceptable because the **DIFFERENCE** between **(ADD1 J)** and **I** decreases in each recursion. That is

Theorem.

```
(IMPLIES (NOT (LESSP J I))
          (ORD-LESSP (DIFFERENCE (ADD1 J)
                                   (ADD1 I))
                     (DIFFERENCE (ADD1 J)
                                   I))))
```

2.7. Recursive Functions on Conventional Data Structures

Because of the need to prove termination, recursive functions on lists generally terminate on an **NLISTP** check (“not **LISTP**”) rather than the **NULL** check conventionally used in Lisp. It is convenient to think of **(NLISTP X)** meaning “**X** is empty.”

Let’s define now another common operation on lists, namely, concatenation.

Definition.

```
(APPEND L1 L2)
=
(IF (NLISTP L1)
    L2
    (CONS (CAR L1)
          (APPEND (CDR L1) L2))))
```

That is, to **APPEND** two lists, **L1** and **L2**, consider whether **L1** is empty or not. If it is empty, the answer is **L2**. Otherwise, **(CAR L1)** is the first element of **L1**, and **(CDR L1)** is the rest. The concatenation of **L1** and **L2** in this case is the list whose first element is **(CAR L1)** and whose remaining elements are obtained by recursively concatenating **(CDR L1)** to **L2**. This definition is admissible because **(COUNT (CDR L1))** is smaller than **(COUNT L1)** if **L1** is a **LISTP**.

Here are some examples of **APPEND**:

```
(APPEND NIL '(1 2 3)) = '(1 2 3)

(APPEND '(1 2 3) '(4 5 6)) = '(1 2 3 4 5 6)

(APPEND '(1 2 3 . ABC) '(4 5 6)) = '(1 2 3 4 5 6)
```

Recall the ‘association list’ concept, as illustrated by

```
'((ADD1 . 1)
  (DIFFERENCE . 2)
  (PLUS . 2)
  (SUB1 . 1)
  (TIMES . 2)
  (ZERO . 0))
```

Here is the function that retrieves the first key-value pair with a given key:

Definition.

```
(ASSOC KEY ALIST)
=
(IF (NLISTP ALIST)
  F
  (IF (EQUAL KEY (CAAR ALIST))
    (CAR ALIST)
    (ASSOC KEY (CDR ALIST)))))
```

The termination argument is the same as that above.

Letting **alist** be the association list shown above, here are some examples of **ASSOC**:

```
(ASSOC 'PLUS alist) = '(PLUS . 2)

(ASSOC 'ZERO alist) = '(ZERO . 0)

(ASSOC 'CUBE alist) = F
```


The following function flattens a binary tree

Definition.

```
(FLATTEN X)
=
(IF (NLISTP X)
  (LIST X)
  (APPEND (FLATTEN (CAR X))
    (FLATTEN (CDR X))))
```

Note that **COUNT** decreases in both recursive calls.

Here are some examples of **FLATTEN**:

```
(FLATTEN 2) = '(2)

(FLATTEN '((1 . 2) (3 . 4) . 5)) = '(1 2 3 4 5)
```

Finally, let us illustrate how we deal with the prohibition of mutual recursion. First, what is mutual recursion? We say **f** and **g** are defined *mutually* recursively if the definition of **f** involves calls of **f** and **g**, and the definition of **g** involves calls of **f** and **g**. This might be written

Definitions.

```
(f x) =  $\phi$ ((f (a x)), (g (b x)))
```

```
(g x) =  $\gamma$ ((f (c x)), (g (d x))).
```

Observe that if we define **f** first and then **g**, then by the time the definition of **g** is presented, **g** is no longer “new;” it is involved in the axiom for **f**. Thus, restriction (1) of the definitional principle is violated, and **g** cannot be defined. The problem is not merely syntactic: the admission of **f** must necessarily involve the admission of **g** and *vice versa*. How then do we make mutually recursive definitions?

The problem can be overcome by a well-known technical trick: define a function **fg** which computes either **f** or **g** according to an auxiliary argument, **flg** (pronounced “flag”):

Definition.

```
(fg flg x)
=
(IF (EQUAL flg 'f)
   $\phi$ ((fg 'f (a x)), (fg 'g (b x)))
   $\gamma$ ((fg 'f (c x)), (fg 'g (d x)))).
```

Note that $(fg \text{ 'f x})$ is $(f \text{ x})$ and $(fg \text{ 'g x})$ is $(g \text{ x})$. The admission of fg will require justifying the intertwined recursion of f and g . This may look like an unattractive solution that could be cured by some syntactic sugar that hides fg from the user. We have found such sugar to be unhealthy. When theorems about f are proved by induction, the induction hypotheses often involve g and *vice versa*. But how can an inductive instance of a theorem about f give a hypothesis about g ? It can't, unless the "theorem about f " is actually a general theorem about fg . That is, the formal statement of sufficiently general inductively provable conjectures about mutually recursive functions frequently involve explicit mention of the intertwined function.

To illustrate the removal of mutual recursion in a concrete example, let us consider the function that substitutes a value for a variable into an expression. We want a function, **SUBSTITUTE**, that when called upon to substitute, say, $'(CAR \text{ L})$ for $'X$ in $'(PLUS \text{ X } (ADD1 \text{ Y}))$, will produce $'(PLUS (CAR \text{ L}) (ADD1 \text{ Y}))$.

Suppose we are substituting some new expression, **new**, for some variable, **var**, into some term **x**. If **x** is a variable, then the answer is **new** if **x** is **var** and is **x** otherwise. If **x** is not a variable it is a list whose **CAR** is the "function symbol" and whose **CDR** is a list of argument terms. We wish to leave the function symbol alone and substitute **new** for **var** into each of the elements of the argument list. But the argument list is arbitrarily long, so we must use recursion to describe the process: if the argument list is empty, return the empty list; otherwise, **SUBSTITUTE new** for **var** into the first argument and cons that onto the result of recursively substituting **new** for **var** into the rest of the argument list. Let us call **SUBSTITUTE-LIST** the function that **SUBSTITUTES** into each term in the argument list. Then we have a mutually recursive pair of functions, **SUBSTITUTE** calls **SUBSTITUTE-LIST** and **SUBSTITUTE-LIST** calls **SUBSTITUTE** and **SUBSTITUTE-LIST**. (Actually, since **SUBSTITUTE** does not call itself directly, we could unfold the definition of **SUBSTITUTE** in the body of **SUBSTITUTE-LIST** and obtain a recursion entirely in terms of **SUBSTITUTE-LIST**; but we prefer the solution described here.)

Following the paradigm in which we introduced **f** and **g** by defining **fg**, we now introduce **SUBSTITUTE** and **SUBSTITUTE-LIST** by defining the single function **SUBST**. **SUBST** takes a "flag" argument, **FLG**, which is equal to **'LIST** if it is playing the role of **SUBSTITUTE-LIST** and **T** if it is playing the role of **SUBSTITUTE**:

Definition.

```
(SUBST FLG NEW VAR X)    ;Substitute NEW for VAR in X.
=
(IF (EQUAL FLG 'LIST)
```

```
; If FLG is 'LIST, X is a list of terms and we substitute
; NEW for VAR in each element of X and return the resulting
; list of terms.
```

```
  (IF (NLISTP X)
      NIL
      (CONS (SUBST T NEW VAR (CAR X))
             (SUBST 'LIST NEW VAR (CDR X)))))
; Otherwise, X is a term.
```

```
  (IF (NLISTP X)
      ; If X is not a list, it is a variable symbol. Is it VAR?
```

```
        (IF (EQUAL X VAR)    ; If so,
            NEW                ; return NEW;
            X)                ; otherwise, return X.
; Otherwise, X is the application of its CAR to the list
; of terms in its CDR. Substitute into each of the argument
; terms with SUBST 'LIST and put the function symbol on
; the front.
```

```
  (CONS (CAR X)
        (SUBST 'LIST NEW VAR (CDR X)))))
```

The function is admitted because the **COUNT** of **X** decreases in each recursive call.

Here are some examples of **SUBST**:

```
(SUBST T      'A 'V 'V)          = 'A
(SUBST T      'A 'V 'X)          = 'X
(SUBST 'LIST 'A 'V '(V X))       = '(A X)
(SUBST T      'A 'V '(PLUS V X)) = '(PLUS A X)
```

2.8. Ordinals

Using natural numbers and lists we represent the ordinals up to ε_0 . ε_0 is the smallest “epsilon number,” i.e., the least ordinal α such that ω^α is α .

Table 2.4 illustrates our representation.

Table 2.4

ordinal	representation
0	0
1	1
2	2
3	3
...	...
ω	'(1 . 0)
$\omega+1$	'(1 . 1)
$\omega+2$	'(1 . 2)
...	...
2ω	'(1 1 . 0)
$2\omega+1$	'(1 1 . 1)
...	...
ω^2	'(2 . 0)
...	...
$\omega^2+\omega+3$	'(2 1 . 3)
...	...
ω^3	'(3 . 0)
...	...
ω^ω	'((1 . 0) . 0)
...	...

Let $\mu(\mathbf{x})$ denote the ordinal represented by \mathbf{x} . If \mathbf{x} is a natural number, $\mu(\mathbf{x})$ is \mathbf{x} . Suppose \mathbf{x} is of the form $'(\mathbf{x}_1 \dots \mathbf{x}_n . \mathbf{k})$ and represents an ordinal. Then $\mu(\mathbf{x})$ is

$$[\sum_{1 \leq i \leq n} \omega^{\mu(\mathbf{x}_i)}] + \mathbf{k}.$$

For \mathbf{x} to represent an ordinal, either \mathbf{x} is a natural number, or \mathbf{x} is of the form $'(\mathbf{x}_1 \dots \mathbf{x}_n . \mathbf{k})$, none of the \mathbf{x}_i s are 0, each represents an ordinal, and $\mu(\mathbf{x}_1) \geq \mu(\mathbf{x}_2) \geq \dots \geq \mu(\mathbf{x}_n)$.

A constructive development of the ordinals up to ε_0 very close to ours may be found in [56], a proof by Gentzen of the consistency of elementary number

theory. His proof uses induction up to ε_0 . Thus, by Gödel's incompleteness theorem, we know that induction up to ε_0 cannot be justified in elementary number theory.

The following functions are defined to compare and recognize ordinals:

- (**LESSP** **I** **J**) If **I** and **J** are both natural numbers, return **T** or **F** according to whether $I < J$.
- (**ORDINALP** **X**) Return **T** or **F** according to whether **X** is the representation of an ordinal less than ε_0 .
- (**ORD-LESSP** **X** **Y**)
If **X** and **Y** are ordinals then return **T** if **X** is smaller than **Y** and **F** otherwise.

Our principles of induction and recursive definition explicitly require certain theorems to be proved about **ORD-LESSP**, establishing that certain quantities decrease in the induction or recursion. In a suitable theory of sets, the soundness of our induction and definition principles can be proved from the assumption that on the **ORDINALP**s **ORD-LESSP** is well-founded, i.e., there is no infinite sequence o_0, o_1, o_2, \dots with the property that for each natural number **i**, (**ORDINALP** o_i) and (**ORD-LESSP** o_{i+1} o_i).

The well-founded lexicographic relation on n-tuples of natural numbers induced by **LESSP** can be obtained by an appropriate use of **ORD-LESSP**. For example, suppose that i_1, j_1, i_2 , and j_2 are all **NUMBERPs**. Then the pair $\langle i_1, j_1 \rangle$ is lexicographically smaller than $\langle i_2, j_2 \rangle$ precisely when

$$(\text{ORD-LESSP } (\text{CONS } (\text{CONS } (\text{ADD1 } i_1) 0) j_1) \\ (\text{CONS } (\text{CONS } (\text{ADD1 } i_2) 0) j_2)).$$

(The **ADD1**s above are present to ensure that the **CONSES** produce ordinals.)

2.9. Useful Functions

We now introduce a variety of useful functions. These functions are part of the basic theory either because (a) they are used in our implementation of the interpreter (e.g., **ASSOC**), (b) we have found it necessary, from a practical point of view, to build knowledge of them into the theorem prover (e.g., **DIFFERENCE** is used in the linear arithmetic decision procedure), or (c) the von Neumann machine on which the theorem prover runs provides means of computing the functions that are significantly faster than merely compiling the recursive definitions (e.g., **QUOTIENT**). Our interest in computational efficiency stems from our desire that the logic be a useful functional programming language rather than from theorem proving considerations alone.

2.9.1. Boolean Equivalence

(**IFF** *P* *Q*) If *P* and *Q* are propositionally equivalent, (i.e., both are **F** or both are non-**F**) return **T**; otherwise, return **F**.

2.9.2. Natural Number Arithmetic

(**FIX** *X*) If *X* is a natural number, return *X*; otherwise return 0.

Functions which “expect” their arguments to be natural numbers usually use **FIX** to “coerce” them.

In all of the function descriptions below, all arguments are coerced with **FIX**.

(**ZEROP** *I*) Return **T** if *I* is 0; otherwise return **F**. Note: Since the argument is coerced with **FIX**, this function returns **T** on non-**NUMBERPs**, e.g., (**ZEROP** **NIL**) = **T**.

(**LESSP** *I* *J*) Return **T** if *I* < *J*; else return **F**. Note: (**LESSP** **T** 4) = **T**.

(**GREATERP** *I* *J*) Return **T** if *I* > *J*; else return **F**.

(**LEQ** *I* *J*) Return **T** if *I* ≤ *J*; else return **F**.

(**GEQ** *I* *J*) Return **T** if *I* ≥ *J*; else return **F**.

(**MAX** *I* *J*) Return the larger of *I* and *J*.

(**PLUS** *I* *J*) Return *I*+*J*. Note: (**PLUS** **T** **F**) = 0.

(**DIFFERENCE** *I* *J*)
If *I* ≥ *J*, return *I*-*J*; else return 0. In number theory this function is sometimes called “monus” or “nonnegative difference.”

(**TIMES** *I* *J*) Return *I***J*.

(**QUOTIENT** *I* *J*) Return the integer quotient of *I* divided by *J*. For example, (**QUOTIENT** 9 4) = 2. If *J* is 0, return 0.

(**REMAINDER** *I* *J*)
Return *I* mod *J*. For example, (**REMAINDER** 9 2) = 1. If *J* is 0, return *I*.

2.9.3. List Processing

(NLISTP X) Return **T** if **X** is not a **LISTP**; otherwise return **F**.

(APPEND L1 L2) Concatenate the lists **L1** and **L2**. For example,

```
(APPEND '(A B C) '(1 2 3))
=
'(A B C 1 2 3).
```

(MEMBER X L) Return **T** if **X** is an element of **L**; otherwise return **F**.

(UNION L1 L2) Concatenate onto the front of **L2** those elements of **L1** not in **L2** and return the result. For example

```
(UNION '(A B C D) '(A C E F))
=
'(B D A C E F)
```

(ADD-TO-SET X L) If **X** is in **L**, return **L**; otherwise return **(CONS X L)**.

(ASSOC X ALIST) Return the first element of **ALIST** whose **CAR** is **X**, or return **F** if no such pair exists.

(PAIRLIST L1 L2) Pair successive elements of **L1** with those of **L2** and return the list of such pairs. (This function is called **zip** in SASL [138].)

2.10. The Interpreter

Among the functions in the logic is an interpreter capable of determining the value of terms in the logic. Of course, the interpreter does not really operate on terms but rather on constants in the logic that represent terms. We encode variable and function symbols with the corresponding literal atoms. We encode function applications with lists. For example, the encoding of the term **(PLUS X Y)** is the constant **'(PLUS X Y)**. Constants can be written either as function applications or in the **QUOTE** notation. Thus, the term **(CONS 0 1)** may be encoded as **'(CONS (ZERO) (ADD1 (ZERO)))** and as **'(QUOTE (0 . 1))**. We call an encoding of a term a “quotation” of the term.

The interpreter takes as its arguments a (quoted) term and an “environment” which assigns values to (quoted) variable symbols. The environment is an association list. The interpreter determines the value of the term in the environ-

ment. Variable symbols are merely looked up in the environment. Numbers and other non-**LITATOM**, non-**LISTPs** evaluate to themselves. Lists whose **CAR** is **QUOTE** evaluate to the **CADR**. All other lists represent function calls of the function symbol in the **CAR** on the arguments in the **CDR**. To evaluate a call of **fn** on some arguments, the interpreter first determines the values of the arguments and then either applies the function named by **fn** (if **fn** is a primitive) or evaluates the body of **fn** in an environment associating the formal parameters with the corresponding argument values (if **fn** is defined). Arguments of **IF**-expressions are handled “lazily.” No other functions are handled lazily.

The function **SUBRP** answers the question “is **fn** a primitive?” If so, we say **fn** “is a **SUBRP**.” The name **SUBRP** comes from the tradition in Lisp, where the functions defined with assembly language were called “subroutines.” If **fn** is a primitive, the function **APPLY-SUBR** applies it to arguments. Thus,

```
(APPLY-SUBR 'ADD1 (LIST 5)) = 6
```

```
(APPLY-SUBR 'CONS (LIST 1 2)) = '(1 . 2)
```

If a symbol is not a **SUBRP** we call it a “non-**SUBRP**.” Non-**SUBRPs** are functions defined by the user with the definitional principle. For such a function name, the functions **FORMALS** and **BODY** return the formal parameters and the body expression of the definition.

For example, let **FINAL-CDR** be the function that returns the “final **CDR**” of a list, i.e., the non-**LISTP** in the rightmost branch of the binary tree:

Definition.

```
(FINAL-CDR X)
=
(IF (LISTP X) (FINAL-CDR (CDR X)) X).
```

Then

```
(SUBRP 'FINAL-CDR) = F
```

```
(FORMALS 'FINAL-CDR) = '(X)
```

and

```
(BODY 'FINAL-CDR) = '(IF (LISTP X)
                           (FINAL-CDR (CDR X))
                           X).
```

The four functions **SUBRP**, **APPLY-SUBR**, **FORMALS**, and **BODY** are undefined on function symbols that have not yet been introduced into the logic. We explain this remark by considering **SUBRP** alone. Logically speaking, there are no axioms characterizing **(SUBRP 'FOO)** if the user has not introduced the

function symbol **FOO**. The expression can be proved neither **T** nor **F**. If one tries to evaluate the expression in the execution environment the attempt fails, as would an attempt to evaluate an expression using **FOO** as a function symbol. Should the user invoke the shell principle to introduce **FOO** as a function symbol, an “extra” axiom is added stating that (**SUBRP 'FOO**) is **T**. On the other hand, should the user invoke the definitional principle to define **FOO**, the “extra” axiom states that (**SUBRP 'FOO**) is **F**. In either case, evaluating the expression in the execution environment will then return **T** or **F** as appropriate. The other three functions, **APPLY-SUBR**, **FORMALS**, and **BODY**, are extended similarly when new functions are introduced.

We now consider formalizing the interpreter for the logic. The task is a subtle one if the consistency of the logic is to be maintained. Just as a naive formalization of recursive definition leads to inconsistency, so too does a naive formalization of functional arguments and embedded interpreters. By creatively instantiating functional arguments, one can lead simply defined interpreters down circular paths that require objects satisfying such inconsistent statements as **X=(ADD1 X)**.

For example, suppose the interpreter is called **EVAL** and that it takes two arguments. Consider the function **RUS**, named in honor of Bertrand Russell, defined as

Definition.

(RUS) = (EVAL '(ADD1 (RUS)) NIL)

This definition is admissible under our current restrictions, since **RUS** is not even recursive. (Readers troubled by the fact that the function **RUS** knows about the literal atom (**PACK '(82 85 83 . 0)**), otherwise written **'RUS**, should consider deriving similar problems without using that atom explicitly.)

Under very plausible assumptions about **EVAL** it is possible to derive such contradictions as

(EVAL '(RUS) NIL) = (ADD1 (EVAL '(RUS) NIL)).

To avoid inconsistency, we axiomatize the interpreter to return either **F** or a pair consisting of the value **v** of the given expression, **x**, and the cost, **c**, of obtaining it. The cost is the number of function applications involved in the evaluation of **x**. More precisely, the cost of evaluating a variable or **QUOTED** form is **0**, the cost of evaluating an **IF** expression is one plus the cost of the test plus the cost of the appropriate branch, the cost of evaluating a call of a **SUBRP** is one plus the cost of evaluating the arguments, and the cost of evaluating a non-**SUBRP** is one plus the cost of evaluating the arguments plus the cost of evaluating the body. The interpreter returns **F** if the cost of evaluating the expression is “infinite.”

Suppose that we have a computer program that evaluates a term in the logic but which operates under a specified bound on the number of function applications permitted in any attempted evaluation. Suppose the program causes an error and halts if the bound is exceeded and otherwise returns the value of the term. Then our interpreter has the following property: if our interpreter returns the pair $\langle v, c \rangle$ on some term x , then the computer program for evaluating terms would return v provided the bound is c or greater; if our interpreter returns F on x then there is no bound that suffices for the computer program to compute x .

Mathematically speaking, our interpreter is a partial recursive function, not a total recursive function. It could not be introduced under our definitional principle. However, the logic contains an axiom that characterizes it and which is satisfied by exactly one function, namely the one described above. In addition, the interpreter is partially implemented in the execution environment of the system. It is possible to apply the interpreter to **QUOTED** terms and, sometimes, get back the value and cost pair. Sometimes however the implemented interpreter runs forever. Nevertheless, it is generally possible to *prove* in those cases that the value of the interpreter is F . (We say “generally” only because sufficiently rich formal systems contain truths which cannot be proved.)

The name of our interpreter is **V&C\$**. “**V&C**” stands for “**Value and Cost**.” The dollar sign is affixed to make it easy to find references to the symbol in our theorem prover’s source code. We will give the function a more permanent name when we are convinced it is a practical way to proceed.

Recall that when we defined the illustrative function **SUBST** we saw the need for two mutually recursive functions: one to process a single term and another to process a list of terms. The first uses the second to process the list of argument terms in a function application, and the second uses the first to process the individual terms in the list. Since **V&C\$** also explores terms, it too is most naturally defined with mutual recursion. We avoid explicit mutual recursion the same way we did with **SUBST**, namely by adding an additional argument which specifies whether the function is processing a single term or a list of terms.

We now illustrate **V&C\$**. In the examples we use the notation $vc \oplus n$ to denote the operation that increments the cost of a value-cost pair, vc , by n and returns F if vc is F .

Suppose **FINAL-CDR** is defined as shown above.

```
(V&C$ T '(FINAL-CDR 'ABC) NIL)
=
(V&C$ T (BODY 'FINAL-CDR) '((X . ABC)))⊕1
=
(V&C$ T '(IF (LISTP X)
              (FINAL-CDR (CDR X))
              X)
          '((X . ABC)))⊕1
```

```

=
(V&C$ T 'X '((X . ABC)))⊕3
=
'(ABC . 3)

```

That is, `(FINAL-CDR 'ABC)` evaluates to `'ABC` and requires 3 function calls, one to expand `FINAL-CDR` into its body, one to evaluate the `IF`, and one to evaluate the `LISTP` test in the `IF`.

Here are some other examples of `V&C$`:

```

(V&C$ T '(FINAL-CDR '(1 . ABC)) NIL)
=
'(ABC . 7)

(V&C$ T '(FINAL-CDR '(1 2 . ABC)) NIL)
=
'(ABC . 11)

```

Note that each time `FINAL-CDR` recurses, the cost is incremented by 4: one to evaluate the `IF`, one to evaluate the `LISTP` test, one to evaluate the `CDR` in the recursive call, and one to enter the body of `FINAL-CDR` again.

Now suppose for the sake of illustration, that we arranged for `RUS` to be defined so that

```

(FORMALS 'RUS) = NIL

(BODY 'RUS) = '(ADD1 (RUS)).

```

Then if we tried to execute `(V&C$ T '(RUS) NIL)` the computation would “run forever”—actually, it would abort after exhausting the available stack space on the host machine. No value would be delivered. However, it is possible to *prove*

```

(V&C$ T '(RUS) NIL) = F.

```

The proof follows quickly from the observation that if `V&C$` returns a pair, `<v, c>`, then `c`, the cost of evaluating `'(RUS)`, is two greater than `c`, the cost of evaluating `'(RUS)`. (The cost goes up by 2 in each recursion, one for entering the definition of `RUS` again and one for the `ADD1`.)

Here then are the functions that are involved in the definition of the interpreter:

```

(SUBRP X)      Return T or F according to whether X is the name of a
                SUBRP.

(APPLY-SUBR FN ARGS)
                If FN is the name of a SUBRP that takes n arguments, apply

```

that function to the first n elements of **ARGS** and return the result. If **ARGS** does not have n elements, extend it on the right with **0s**.

(FORMALS FN) If **FN** is the name of a non-SUBRP, return the list of formal parameters from the definition.

(BODY FN) If **FN** is the name of a non-SUBRP, return the s-expression that is the body of the definition.

(V&C-APPLY\$ FN ARGS)

FN is assumed to be the name of a function (a SUBRP or non-SUBRP) of arity n and **ARGS** is assumed to be a list of n “value-cost pairs.” Some “pairs” may be **F**. Apply **FN** to the value components of the pairs and return the corresponding value-cost pair or **F** if either some “pair” was **F** or **FN** fails to terminate. However, a special case is made if **FN** is **IF**: the answer is **F** only if the test or the appropriate branch “pair” is **F**; the termination of the branch not selected by the test is irrelevant.

(V&C\$ FLG X ALIST)

If **FLG** is **T** (or any value other than **'LIST**), **X** is treated as a term and evaluated in the environment **ALIST**. If it can be evaluated with a finite cost, the answer returned is the pair containing the value of **X** and the cost. If it cannot be evaluated with a finite cost, the answer returned is **F**. (The implemented version of **V&C\$** never returns **F** but runs forever instead.) If **FLG** is **'LIST**, **X** is treated as a list of terms, each is evaluated in the environment **ALIST**, and the list of results is returned.

It is convenient to have a version of the interpreter that deals just with values and not value-cost pairs. We define the following two functions which are analogous to **V&C-APPLY\$** and **V&C\$**:

(APPLY\$ FN ARGS)

FN is assumed to be a function name (a SUBRP or non-SUBRP), and **ARGS** is a list of values. Apply **FN** to **ARGS** and return the result. **APPLY\$** is defined to be just the **CAR** of the corresponding **V&C-APPLY\$**.

(EVAL\$ FLG X ALIST)

Return the value of **X** in the environment **ALIST**, treating **X** either as a single term or a list of terms, depending on **FLG** as in **V&C\$**. **EVAL\$** computes its answer by **APPLY\$**ing each function symbol in the term to the recursively obtained values of the argument terms.

Since **APPLY\$** is defined to be the **CAR** of **V&C-APPLY\$**, it can be shown that **APPLY\$** returns **0** when **V&C-APPLY\$** returns **F**. That is, if the application of **FN** to **ARGS** provably fails to terminate, **APPLY\$** provably returns **0**. Of course, **APPLY\$**, like **V&C-APPLY\$**, will run forever on some executions.

EVAL\$ is *not* defined to be the **CAR** of the corresponding **V&C\$** but instead uses **APPLY\$** to apply each function symbol in the term to the recursively obtained values of the arguments. For example, using **V&C\$** to evaluate **'(PLUS (RUS) 4)** produces **F**, since **'(RUS)** does not terminate. But

```
(EVAL$ T '(PLUS (RUS) 4) NIL)
=
(PLUS (EVAL$ T '(RUS) NIL)
      (EVAL$ T '4 NIL))
=
(PLUS (APPLY$ 'RUS NIL) 4)
=
(PLUS 0 4)
=
4.
```

EVAL\$ on **'(PLUS (RUS) 4)** will not evaluate to **4**—it runs forever. However, it can be proved that the **EVAL\$** expression is equal to **4**.

Our definition of **EVAL\$** is attractive because for functions which always terminate, such as **PLUS**, we get theorems of the form:

```
(EQUAL (EVAL$ T (LIST 'PLUS X Y) A)
      (PLUS (EVAL$ T X A)
            (EVAL$ T Y A)))
```

The advantage of this will become apparent later.

EVAL\$ and **V&C\$** are extremely powerful programming tools as they permit one to pass around “terms” as objects in the language. One can define many useful general purpose functions using **EVAL\$** or **V&C\$**.

For example, the following function sums values of the expression **EXP** as the variable **V** takes as its value the successive elements of **LST**:

Definition.

```
(SIGMA V LST EXP)
=
(IF (LISTP LST)
  (PLUS (EVAL$ T EXP (LIST (CONS V (CAR LST))))
        (SIGMA V (CDR LST) EXP))
  0)
```

Note that (a) this function is not total (although we compute its value in all the cases in which it terminates), but (b) this definition is acceptable.

Here is an example of **SIGMA** to compute the sum, as **I** ranges from 1 to 3, of **I**²:

```
(SIGMA 'I '(1 2 3) '(TIMES I I))
=
(PLUS (TIMES 1 1)
      (PLUS (TIMES 2 2)
            (PLUS (TIMES 3 3)
                  0)))
=
14.
```

In addition to being useful as a programming aid, many beautiful theorems can be proved about **SIGMA**. For example

```
(EQUAL (SIGMA V (APPEND L1 L2) EXP)
      (PLUS (SIGMA V L1 EXP)
            (SIGMA V L2 EXP)))

(EQUAL (SIGMA V L (LIST 'PLUS EXP1 EXP2))
      (PLUS (SIGMA V L EXP1)
            (SIGMA V L EXP2)))
```

It is here that one of the main attractions of our **EVAL\$** is apparent. Had we defined **EVAL\$** to be the **CAR** of **V&C\$** the latter formula would not be a theorem—on the left-hand side, the nontermination of **EXP1** would prevent **EXP2** from making its contribution to the sum, while on the right-hand side, **EXP2** makes its contribution. But arranging for the latter formula to be a theorem permits the convenient manipulation of quantified expressions.

EVAL\$ and **V&C\$** can also be used to define and investigate the properties of partial functions. For example, suppose we arranged for

```
(FORMALS 'LEN) = '(L)

(BODY 'LEN) = '(IF (NULL L)
                  (QUOTE 0)
                  (ADD1 (LEN (CDR L)))).
```

Recall that the specious acceptance of a similar “definition” earlier led to a contradiction.

However, by evaluating **'(LEN '(1 2 3))** with **EVAL\$** we can obtain the desired answer 3. And we can *prove* that the **V&C\$** of **'(LEN '(1 2 3 . ABC))** is **F**. That is, the function doesn't terminate on a list ending with **'ABC**. We can indeed prove the general result that **LEN** terminates iff the argument list ends with **NIL**.

How can we arrange for the **FORMALS** and **BODY** of **'LEN** to be as assumed above? We make a special provision for this by recognizing when a function has been defined by a top-level call to **EVAL\$** on a **QUOTED** body and an environment that passes the formals in unaltered. Thus, the **FORMALS** and **BODY** of **'LEN** are as shown above if **LEN** is defined with

Definition.

```
(LEN L)
=
(EVAL$ T '(IF (NULL L)
               (QUOTE 0)
               (ADD1 (LEN (CDR L)))))
  (LIST (CONS 'L L))).
```

We are the first to admit the hackish nature of this mechanism of defining **FORMALS** and **BODY** when the body used in the definition is such an **EVAL\$** expression. The rationale behind our choice of this mechanism follows.

First, note that unlike the mere assumptions that the **FORMALS** and **BODY** of **'LEN** are as desired, this mechanism actually introduces the function symbol **LEN** into the logic. Not only can we talk about the result of interpreting expressions such as **'(LEN X)** with **EVAL\$** or **V&C\$**, but we can talk about such terms as **(LEN X)**. In the event that the recurrence implicit in the computation terminates, the value of **'(LEN X)** obtained by interpretation (under the appropriate assignment to **'X**) is in fact **(LEN X)**. That is, if **X** ends in a **NIL**, then

```
(EVAL$ T '(LEN X) (LIST (CONS 'X X)))
```

is **(LEN X)**.

However, matters are confusing if the computation fails to terminate. The **V&C\$** of **'(LEN '(1 2 3 . ABC))** is provably **F**. But the **EVAL\$** of **'(LEN '(1 2 3 . ABC))** is provably 0—because the **APPLY\$** of **'LEN** to **'(1 2 3 . ABC)** is 0. However, the term **(LEN '(1 2 3 . ABC))** is provably equal to 1 (because the **EVAL\$** in the defining axiom for **'LEN** **APPLY\$**s **ADD1** to the **APPLY\$** of **'LEN** to **'(2 3 . ABC)**, which is 0). Meanwhile, the evaluation of all three of these expressions in the execution environment runs forever.

In [30] we explain the formal axiomatization of **V&C\$** at length, and we show several termination and nontermination proofs using **V&C\$**.

2.11. The General Purpose Quantifier

The function **SIGMA**, shown above, is only one of many useful “quantifiers.” In addition to summing the value of some term over a list, we may wish to take the product, check that the term is true on every element or on some element, find the maximum, collect the successive values into a new list, etc. We call all of these concepts “quantifiers” because they involve the notion of a “bound” variable which takes on the values in some specified domain. Using **EVAL\$** we define a general purpose bounded quantifier function, called **FOR** and modeled on Teitelman’s iterative statement, **FOR**, in Interlisp [137].

The term

```
(FOR 'I L '(MEMBER I S)
      'SUM '(TIMES I I)
      (LIST (CONS 'S S)))
```

may be read “for **I** in **L** when **I** is member of **S** sum **I**².” The final argument to the **FOR** is an association list that maps from the “free variables” in the quantified expressions to their values. In this case, the alist maps from the **LITATOM 'S** to the variable **S**.

```
(FOR VAR LST COND OP BODY ALIST)
```

In common use, **VAR** is a **LITATOM** denoting a variable, **LST** is a list, **COND** and **BODY** are both the quotations of terms, **OP** is one of the keywords below indicating the operation to be performed, and **ALIST** is an association list. The definition of **FOR** is recursive: if **LST** is empty, **FOR** returns an “initial value” that is determined by **OP**. Otherwise **LST** is some element **e** followed by some remaining elements **tl**. **FOR** binds **VAR** in **ALIST** to **e** and evaluates the **COND** expression in that environment. If the value of the **COND** expression is non-**F**, **FOR** evaluates **BODY** and performs the indicated “quantifier operation” on the value of **BODY** and the result of recursing on **tl**. Otherwise, if the value of the **COND** expression is **F**, **FOR** does not evaluate **BODY** and returns the result of recursing on **tl**.

The permitted operations and the values of the corresponding **FOR** statements are as follows:

ADD-TO-SET	Return the list of successive evaluations of BODY , with duplicates eliminated.
ALWAYS	Return T if every evaluation of BODY returns non- F ; otherwise return F .

APPEND	Return the concatenation of the successive evaluations of BODY .
COLLECT	Return the list of successive evaluations of BODY .
COUNT	Return the number of times BODY evaluated to non- F .
DO-RETURN	Return the first evaluation of BODY —that is, return the value of BODY on the first element of the range for which COND evaluates to non- F .
EXISTS	Return T if there is an element of the range on which BODY evaluates to non- F .
MAX	Return the maximum of the evaluations of BODY .
SUM	Return the sum, as with PLUS , of the evaluations of BODY .
MULTIPLY	Return the product, as with TIMES , of the evaluations of BODY .
UNION	Return the list obtained by unioning together the successive evaluations of BODY .

The syntax contains some syntactic sugar for writing **FOR** expressions. While the function **FOR** takes six arguments we permit “applications” of **FOR** to both five and seven “arguments.” In such ill-formed applications, some of the “arguments” are “noise” words. Others are terms which are replaced by their quotations. In addition, the five and seven argument versions omit the association list mapping the quotation of free variables in the quantified expressions to their values.

An example of the seven argument use of **FOR** is

```
(FOR X IN L
  WHEN (LESSP X 100)
  SUM (TIMES A B X)).
```

This abbreviates

```
(FOR 'X L
  '(LESSP X (QUOTE 100))
  'SUM '(TIMES A (TIMES B X))
  (LIST (CONS 'A A)
        (CONS 'B B)))
```

The five argument use of **FOR** permits one to drop the **WHEN** keyword and the following condition in situations when the condition is **T**.

In [30] we carefully explain our axiomatization of **FOR** and some of the alternative methods we considered. In addition, we list many “schematic” theorems proved about **FOR**.

The function **EXPLODE**, defined below, is used in Table 2.5 to illustrate **FOR** statements and their values.

Definition.

```
(EXPLODE WRD)
=
(FOR X IN (UNPACK WRD) COLLECT (PACK (CONS X 0))).
```

Table 2.5

term	value
(EXPLODE 'TREE)	= '(T R E E)
(FOR X IN '(CAT DOG MAN OX POT) = '(DOG OX POT) WHEN (MEMBER 'O (EXPLODE X)) COLLECT X)	
(FOR X IN '(CAT HIT HEN) COLLECT (EXPLODE X))	= '((C A T) (H I T) (H E N))
(FOR X IN '(CAT HAT DOG) APPEND (EXPLODE X))	= '(C A T H A T D O G)
(FOR X IN '(CAT HOT DOG) UNION (EXPLODE X))	= '(C A H T D O G)
(FOR X IN '(CAT HAT MAT) ALWAYS (MEMBER 'A (EXPLODE X))	= T
(FOR X IN '(CAT HAT HEN) WHEN (MEMBER 'H (EXPLODE X)) DO-RETURN (CADR (EXPLODE X)))	= 'A

2.12. Introducing Functions by Constraint

It is sometimes useful to introduce new function symbols with certain properties without defining the new symbols. To motivate this idea, consider the definition by Goldschlag in [60] of an operational semantics for a simple concurrent programming language. The definition relies upon the notion of a “scheduler” that selects which of the statements in a nonempty program, **PRG**, will be executed at time **I**. This scheduler function, named **CHOOSE**, is not defined but is merely assumed to have several properties, one of which is that if **PRG** is a **LISTP** then **(CHOOSE PRG I)** is an element of **PRG**. Certain theorems about the programming language are then derived from the assumed properties of **CHOOSE** and the operational semantics. Because **CHOOSE** is left undefined, it stands to reason that the derived theorems are valid no matter what scheduler is used, as long as it has the properties assumed about **CHOOSE**.

The above example should raise several questions in the reader’s mind. How can a symbol be introduced and constrained to have some properties without defining it? Is this any different from just adding new axioms to the logic? What happens if you try to evaluate such a symbol on particular arguments? Can the informal remark, that “it stands to reason that the derived theorems are valid no matter what scheduler is used,” be made precise? And if so, can those theorems be used effectively to derive properties of the language when particular schedulers are used? We answer these questions here, in the primer. In the next chapter, where we discuss how the logic can be used to overcome common formalization problems, we illustrate the use of constrained functions in a variety of settings.

To introduce the new function symbols \mathbf{f}_1 , ..., and \mathbf{f}_n , and simultaneously constrain them to satisfy the formula **ax**, we write:

Constraint.

Constrain \mathbf{f}_1 , ..., and \mathbf{f}_n to have property **ax**.

Logically speaking, this is equivalent to adding **ax** as an axiom. However, just as the definitional principle enforces restrictions that prevent the addition of an inconsistent axiom, the constraint mechanism has restrictions. The main restriction ensures that the new symbols *could* be defined so that **ax** were a theorem. This is done by requiring the user to exhibit “witness functions” \mathbf{g}_1 , ..., \mathbf{g}_n , that when used in place of \mathbf{f}_1 , ..., \mathbf{f}_n in **ax** produce a formula, **ax’** that is provable.

Although in Goldschlag’s work there are additional constraints on **CHOOSE**, were **CHOOSE** to be constrained only by the above noted requirement, i.e., that it select some element of a nonempty **PRG**, then it could be introduced with

Constraint.

Constrain **CHOOSE** to have property

(**IMPLIES** (**LISTP** **PRG**)
 (**MEMBER** (**CHOOSE** **PRG** **I**) **PRG**)).

To prove admissibility the user must exhibit a witness for **CHOOSE**. A suitable witness is the function that takes two arguments, **PRG** and **I**, ignores **I** and returns (**CAR** **PRG**), called **A-CHOOSE-WITNESS** below.

Definition.

(**A-CHOOSE-WITNESS** **PRG** **I**)
 =
 (**CAR** **PRG**).

If all calls of **CHOOSE** in the constraint above are replaced by calls of **A-CHOOSE-WITNESS** then the resulting formula simplifies, by expanding the definition of the witness, to

(**IMPLIES** (**LISTP** **PRG**)
 (**MEMBER** (**CAR** **PRG**) **PRG**)),

which is a theorem. Note that the witness used here is not “in the spirit” of the intended scheduler, since it selects the same element each time. Such trivial witnesses often suffice and, since the witness functions are irrelevant once the constraint has been shown satisfiable, users should always consider trivial witnesses.

Suppose the user had desired the more powerful choice function described by

Constraint?

Constrain **CHOOSE** to have property

(**MEMBER** (**CHOOSE** **PRG** **I**) **PRG**).

The attempt to introduce such a **CHOOSE** would fail. No witness can be found since no suitable value exists for (**CHOOSE** **NIL** **I**). Had the user added (**MEMBER** (**CHOOSE** **PRG** **I**) **PRG**) as an axiom, without using the constraint mechanism, the resulting logic would be inconsistent since (**MEMBER** **X** **NIL**) = **F** is a theorem.

Once **CHOOSE** has been introduced by constraint it cannot be further constrained. More generally, only new function symbols can be constrained. It is for this reason that the constraint mechanism allows the introduction of many new symbols at once. Often the constraining axiom is a conjunction of several formulas relating the new symbols.

Recall that an execution environment is provided with the mechanized logic. In that environment certain functions can be tested by evaluating them on constants. For example, (**A-CHOOSE-WITNESS** '(**A** **B** **C**) **3**) evaluates to '**A**. Constrained functions cannot be evaluated. Attempting to do so, e.g., by

submitting the expression (**CHOOSE** ' (**A B C**) **2**), results in a simple error message. It may occasionally happen that the axioms can be used to determine the value of such an expression. For example, (**CHOOSE** ' (**A**) **2**) can be *proved* to be '**A**', using only the constraint. But the execution environment always causes an error if an attempt is made to evaluate a constrained function. Defined functions can be evaluated when the evaluation process does not encounter a constrained (or otherwise undefined) function.

Finally, we turn to the intuitive remark suggesting that if a function satisfies the constraint of another then the function enjoys any property proved about the other. Suppose **CHOOSE** is introduced by the admissible constraint above. Then, if (**LENGTH PRG**) is defined as the number of elements in **PRG**, it is possible to prove the following theorem:

Theorem. CHOOSE-IS-CAR:
 (**IMPLIES** (**EQUAL** (**LENGTH PRG**) **1**)
 (**EQUAL** (**CHOOSE PRG I**) (**CAR PRG**))).

This theorem says that **CHOOSE** is **CAR** if its first argument has only one element.

Now suppose we define some particular scheduler, say **CHOOSE-FAIRLY**, that chooses the statements in **PRG** as some function of time so as to guarantee that each statement is chosen infinitely often. Suppose we prove that **CHOOSE-FAIRLY** "is a scheduler," by which we mean **CHOOSE-FAIRLY** satisfies the constraint on **CHOOSE**. Then we claim that **CHOOSE-FAIRLY** is just **CAR** when its first argument has only one element:

Theorem. CHOOSE-FAIRLY-IS-CAR:
 (**IMPLIES** (**EQUAL** (**LENGTH PRG**) **1**)
 (**EQUAL** (**CHOOSE-FAIRLY PRG I**)
 (**CAR PRG**))).

To see why this must be true, consider the proof of **CHOOSE-IS-CAR**. Since **CHOOSE** was a new symbol when it was introduced by constraint, the only axiom about **CHOOSE** is the constraint itself. Since **CHOOSE-FAIRLY** "is a scheduler" we have an analogous theorem about **CHOOSE-FAIRLY**. Thus, we could construct a proof of **CHOOSE-FAIRLY-IS-CAR** from the proof of **CHOOSE-IS-CAR** by simply replacing every appeal to the constraining axiom about **CHOOSE** by an appeal to the analogous theorem about **CHOOSE-FAIRLY**.

While we could actually reconstruct a proof of the new result, it suffices to just show that we always could so construct a proof. Indeed, that is what we do in [19] and the result is a new "derived rule of inference" called *functional instantiation*. Roughly put it says that if **thm** is any theorem and **fs** is a mapping of function symbols to function symbols, and the image under **fs** of every axiom

involved in the proof of **thm** is a theorem, then the image under **fs** of **thm** is a theorem.

Thus, any theorem about **CHOOSE** is also a theorem about any scheduler and can be derived about a particular scheduler **fn** by functional instantiation, at the mere cost of showing that **fn** is indeed a scheduler. This remark is imprecise since, for example, we have not defined what we mean by a theorem “about” a function or how to change a theorem “about” one function into one “about” another. A complicating issue is that the constrained functions might be involved in the definitions of other functions and those definitions must be considered as well. But the intuition that we should be able to derive theorems about particular schedulers directly from the theorems about **CHOOSE** is well founded and its precise formulation merely awaits the development of clear terminology.

3 Formalization Within the Logic

Given the descriptions of the primitive functions in the logic it is easy to see that the value of

(APPEND NIL '(4 5 6))

is **'(4 5 6)**, and the value of

(APPEND '(1 2 3) '(4 5 6))

is

'(1 2 3 4 5 6).

Another way to phrase the last example is that the value of

**(EQUAL (APPEND '(1 2 3) '(4 5 6))
 '(1 2 3 4 5 6))**

is **T**.

Now what is the value of **(APPEND NIL Y)**? In general, we cannot answer such a question unless we know the values of the variables involved, in this case, **Y**. But whatever the value of **Y**, the value of **(APPEND NIL Y)** is the same as that of **Y**. That is,

(EQUAL (APPEND NIL Y) Y)

has the value **T**, regardless of the value of **Y**.

Similarly, the value of

```
(EQUAL (APPEND (APPEND X Y) Z)
        (APPEND X (APPEND Y Z)))
```

is **T**, regardless of the values of **X**, **Y**, and **Z**.

The obvious question is “Can you prove it?” It is easy to *test* these assertions by executing the given expressions under sample assignments to the variables. But is it possible to demonstrate that the value must always be **T**? If all we have to work with is the informal description of the logic as a programming language, as given in the previous chapter, the answer to these questions is “no, we can’t prove it, but would you like to see another test?”

In the next chapter we present the logic again but do so formally. We define the *language* precisely so that you know exactly what is legal to write down. We give *axioms* that completely describe each function, and we give *rules of inference* with which the axioms can be manipulated. A *theorem* is any formula that is either an axiom or can be derived from theorems using the rules of inference. It happens that every theorem is always true, regardless of the values of the variables. More precisely, if **t** is some theorem containing the variables **v**₁, ..., **v**_n, then regardless of what values we assign to **v**₁, ..., **v**_n, the value of **t** in any model of our axioms is non-**F**. This is no accident; the rules of inference are all validity-preserving so any formula constructed from the axioms by the rules of inference is necessarily always true.

So, if for some reason you wish to determine whether some formula has the value **T** in all models, you can try proving it. Technically speaking, this is not always possible. That is, there are formulas that are true in all models but that cannot be proved. That is what Gödel’s incompleteness theorem tells us. But you are extremely unlikely to happen upon a truth that is not a theorem and for all practical purposes you should proceed as though every truth in the logic has a proof.

Why might you want to know whether some formula is always true? Suppose you are writing a “program” (i.e., function) in the logic and you need to compute **(APPEND A (APPEND B C))**. Suppose you have already computed **(APPEND A B)**, and it is the value of the variable **AB**. Then you might use **(APPEND AB C)**, i.e., **(APPEND (APPEND A B) C)**, for **(APPEND A (APPEND B C))**. This is a legitimate thing to do, *if* you know that **APPEND** is associative. That is, if you know that

```
(EQUAL (APPEND (APPEND X Y) Z)
        (APPEND X (APPEND Y Z)))
```

always evaluates to **T**.

The situation just described is a simple example of a very deep problem: *formalization*. When confronted with a practical computing question of the

form “Have I done this correctly?” it is often possible to convert it into a formal question of the form “Is this formula a theorem?” We offer no mechanical assistance in this process; indeed, there is a sense in which no mechanical assistance is possible: the question bridges the gap between the informal and the formal and, by necessity, any mechanical help must confine itself to the formal side of the chasm.³

Almost invariably, when confronted with the challenge to formalize a problem, new users of our logic are baffled—especially if they come from a mathematical background. How can anything as complicated or as abstract as your problem be couched in the simple, nearly constructive system described here? It is hard enough to formalize concepts in a truly powerful mathematical system such as set theory. To do it in a logic with no quantification, no mutual recursion, no infinite objects, no sets, no abstraction, no higher order variables, and a very “constructive” sense of partial functions is, it would seem, impossible. For the authors of the system to then inveigh against the addition of axioms as being too dangerous simply seems to doom the system to total uselessness. These impressions notwithstanding, many complicated and abstract problems have been formalized within the logic. This chapter explains how it is done.

It is not necessary to be able to prove theorems in order to formalize problems. It is necessary to have a clear understanding of what formulas in the logic “mean”—a topic that is often illuminated by the axioms and rules of inference. But we believe the primer gives a description of the language adequate for the discussion of formalization. Furthermore, the discussion of formalization clarifies some of the issues and may motivate some readers to undertake the necessary study of Chapter 4. Readers who disagree may wish to read Chapter 4 first and then return to this chapter.

This chapter is organized as follows. First, we show a few recursive definitions, in the spirit of those offered in the primer, to illustrate the use of the logic as a programming language. However, we basically assume in this book that you are sufficiently familiar with programming to use recursion and inductive data structures to implement solutions to elementary programming problems. Of more concern in this chapter is the expression of relationships and the definitions of concepts that most readers think of as being “mathematical” rather than “computational.” Thus, the second section of the chapter illustrates the formal

³We take the view that anything that is codified in a programming language—be it a low level machine code or a high level language like Prolog—is formal. Researchers trying to bring the informal world within the grasp of computing machines are, in our view, merely formalizing a fragment of that world. Regardless of the extent of their success, there will necessarily be a gap between that which has been formally modeled and that which we personally want or desire or intend of our programs.

statement of some familiar mathematical relationship such as associativity, transitivity, etc. We describe how we deal with some of the omissions from our logic: mutual recursion, unbounded quantification, sets, and infinities. We then deal with two problems uniquely associated with features of our logic (rather than omissions from it), namely the use of **V&C\$** to discuss termination questions and the statement of general purpose “schematic” theorems about **FOR**. Finally, we discuss two commonly occurring formalization problems: nondeterminism and language semantics. The latter problem is discussed in the more general framework of the formalization of other formal systems within ours.

3.1. Elementary Programming Examples

In section 2.5 (page 36) of the primer, we showed how the primitive shells are used to represent sequences, words, tables, and trees. In section 2.7 (page 44), we gave definitions for list concatenation, **APPEND**; table lookup, **ASSOC**; tree flattening, **FLATTEN**; and term substitution, **SUBST**. These are typical examples of uses of the logic as a programming language. They are also examples of formalization. Is “list concatenation” **APPEND**? We cannot answer the question, nor can we tell you how to invent the formal definition of **APPEND** in response to the challenge “Formalize list concatenation.” In lieu of definite answers, we offer more examples. In this section we’ll implement in the logic push down stacks, finite queues, and an insertion sort function.

3.1.1. Push Down Stacks

Problem. Formalize push down stacks on which only numbers are pushed. Since we are dealing with an applicative programming language a stack is an object and the operations on it are functions that return “new” objects with certain functional relations to the input. Informally we want the following five functions:

PUSH	Construct the stack obtained by pushing a given number onto a given stack of numbers.
EMPTY	Return the empty stack of numbers.
TOP	Return the topmost element of a given stack of numbers.
POP	Return the stack obtained by popping the topmost element off a given stack of numbers.
STACKP	Return T or F according to whether the given argument is a stack of numbers.

This informal specification can be formalized by the following invocation of the shell principle:

Shell Definition.

Add the shell **PUSH** of 2 arguments
with base function **EMPTY**,
recognizer function **STACKP**,
accessor functions **TOP** and **POP**,
type restrictions **(ONE-OF NUMBERP)** and **(ONE-OF STACKP)**, and
default functions **ZERO** and **EMPTY**.

We leave to the reader the difficult question “Is that really what is meant by the informal specification?”

Several observations are in order however. Let **STK** be the push down stack constructed by **(PUSH 3 (PUSH 2 (PUSH 1 (EMPTY))))**, that is, the stack obtained by pushing **1**, then **2**, and then **3** onto the empty stack. Then **(TOP STK)** is **3**, the top of the stack. **(POP STK)** is **(PUSH 2 (PUSH 1 (EMPTY)))**, the result of popping the **3**. **(STACKP STK)** is **T**.

Several “pathological” cases are included in the formalization, for better or for worse. **(PUSH T STK)** produces the same stack as **(PUSH 0 STK)** because **T** is not a number and so **PUSH** “coerces” it to the first default object **0**. **(TOP (EMPTY))** is **0** because **TOP** returns the first default object when called on a non-stack or on the empty stack. **(POP F)** is **(EMPTY)** because **POP** returns the second default object when called on a non-stack or the empty stack.

3.1.2. Finite Queues

Problem. Formalize finite queues. Informally, we want the following functions:

MAKE-QUEUE	Return an empty finite queue of a given maximum size.
ENQUEUE	Add a new element to the front of a given finite queue.
DEQUEUE	Given a finite queue, return the finite queue with the last element removed.
ELE	Return the last element of a given finite queue.
EMPTY-QUEUEP	Return T or F according to whether a given finite queue is empty.
FULL-QUEUEP	Return T or F according to whether a given finite queue is full.

QUEUEP Return **T** or **F** according to whether the argument is a finite queue.

The basic idea behind our formalization is that a finite queue is a pair, $\langle \text{list}, \text{max} \rangle$ where **list** is a list of the elements currently in the queue, ordered from oldest to newest, and **max** is the maximum size of the queue. We cannot use the shell principle alone to formalize this approach because we have no way, within the shell principle, to enforce the constraint that the length of **list** is less than or equal to **max**. Therefore, we will use the shell principle to get a “new” kind of ordered pair, called a **PROTO-QUEUE**, and we will formalize queues in terms of it. By ordering **list** as indicated we can formalize **ENQUEUE** by **APPENDING** the new element to the **list**, **DEQUEUE** by taking the **CDR**, and **ELE** by taking the **CAR**.

Here is the formalization of **PROTO-QUEUES**:

Shell Definition.

Add the shell **PROTO-QUEUE** of 2 arguments
with recognizer function **PROTO-QUEUEP**,
accessor functions **QLIST** and **QMAX**,
type restrictions **(NONE-OF)** and **(ONE-OF NUMBERP)**, and
default functions **ZERO**, and **ZERO**.

MAKE-QUEUE is then defined as

Definition.

```
(MAKE-QUEUE MAX)
=
(PROTO-QUEUE NIL MAX).
```

We can define **QUEUEP** to be

Definition.

```
(QUEUEP X)
=
(AND (PROTO-QUEUEP X)
      (LEQ (LENGTH (QLIST X))
            (QMAX X))),
```

where

Definition.

```
(LENGTH X)
=
(IF (NLISTP X)
    0
    (ADD1 (LENGTH (CDR X)))).
```

We can define **EMPTY-QUEUEP** and **FULL-QUEUEP** as follows:

Definition.

```
(EMPTY-QUEUEP Q)
=
(AND (QUEUEP Q) (NLISTP (QLIST Q)))
```

Definition.

```
(FULL-QUEUEP Q)
=
(AND (QUEUEP Q)
      (EQUAL (LENGTH (QLIST Q))
              (QMAX Q))).
```

What should **ENQUEUE**, **DEQUEUE**, and **ELE** do on “illegal” arguments such as non-queues or full queues or empty queues? Do we implement them with “run-time” checks and make them return special answers in “illegal” situations? For example, we could define **ENQUEUE** so that if its argument were not a **QUEUEP** or were full, it returned **F**. If we do not include run-time checks, the functions may return “garbage” in illegal situations. Our own personal bias is to avoid run-time checks, use the functions in a way that is consistent with their intended specifications, and prove that every use is appropriate. Here then are the definitions:

Definition.

```
(ENQUEUE E Q)
=
(PROTO-QUEUE (APPEND (QLIST Q)
                      (LIST E))
              (QMAX Q)).
```

Observe that if **Q** is not a **QUEUEP** or is full, we return a non-**QUEUEP** in which the list component is longer than the maximum size.

Definition.

```
(DEQUEUE Q)
=
(PROTO-QUEUE (CDR (QLIST Q))
              (QMAX Q)).
```

We intend for **DEQUEUE** to be used only on nonempty **QUEUEPs**. If **Q** is not even a **PROTO-QUEUEP**, **DEQUEUE** actually returns the empty **QUEUE** of size 0. If **Q** is a **PROTO-QUEUE** but not a **QUEUEP**—meaning its list component is too long—**DEQUEUE** shortens the list by 1, and the answer may or may not be a **QUEUEP**. If **Q** is a **QUEUEP** but is empty, **DEQUEUE** returns an empty **QUEUEP** of the same size.

Definition.

```
(ELE Q)
=
(CAR (QLIST Q)).
```

Observe that if **Q** is not a **PROTO-QUEUEP** or is the empty **QUEUEP**, **ELE** returns 0. If **Q** is a **PROTO-QUEUEP** that fails to be a **QUEUEP**, **ELE** returns the first element of its list component.

Once again the hard question is left to the reader: “Is that what we mean by finite queues?”

3.1.3. Insertion Sort

Problem. Define a function that sorts lists of numbers into ascending order. Our approach will be to define an insertion sort function.

The function **INSERT** takes a number **N** and an ordered list of numbers **L** and returns the list obtained by inserting **N** into **L** “at the right place,” i.e., immediately before the first element of **L** that is greater than or equal to **N**:

Definition.

```
(INSERT N L)
=
(IF (NLISTP L)
    (LIST N)
    (IF (LEQ N (CAR L))
        (CONS N L)
        (CONS (CAR L) (INSERT N (CDR L)))))).
```

Thus, **(INSERT 5 '(2 4 5 6))** is **'(2 4 5 5 6)**.

The main function, **SORT**, takes a list **L** and recursively sorts its **CDR**, when nonempty, and then **INSERT**s the **CAR** of **L** into the result

Definition.

```
(SORT L)
=
(IF (NLISTP L)
    NIL
    (INSERT (CAR L) (SORT (CDR L)))).
```

Thus, **(SORT '(4 2 5 1 5 6))** is **'(1 2 4 5 5 6)**.

3.2. Elementary Mathematical Relationships

We now turn to the problem of expressing elementary relationships between functions as theorems in the logic. Below are formulas expressing familiar facts of arithmetic:

(EQUAL (PLUS I J) (PLUS J I))	;Commutativity of PLUS
(EQUAL (PLUS (PLUS I J) K) (PLUS I (PLUS J K)))	;Associativity of PLUS
(EQUAL (PLUS I 0) I)	;Right Identity of PLUS
(EQUAL (TIMES X (PLUS I J)) (PLUS (TIMES X I) (TIMES X J)))	;Distributivity of TIMES ;over PLUS
(IMPLIES (AND (LESSP I J) (LESSP J K)) (LESSP I K))	;Transitivity of LESSP

Consider the first formula. The formula is a theorem and thus it is “always true.” That is, no matter what values we choose for **I** and **J**, the sum **I** plus **J** is the same as that of **J** plus **I**. In other words, **PLUS** is a commutative function. In more traditional notation this might be written: $\forall i \forall j \ i+j = j+i$. Such elementary facts of arithmetic are so often expressed formally (i.e., as formulas) that it is not difficult to write them in our notation, once you get used to our prefix syntax and the absence of quantifiers. One way to think of the quantifier situation is that all formulas are (implicitly) universally quantified over all variables on the far outside of the formula.

However, one must be especially careful of the formalization of familiar concepts. One of the five formulas above is not a theorem. Do you know which one?

It is the third. Normally, when we say “**0** is the right identity of plus” we implicitly restrict our attention to numeric arguments. What is the sum of **T** and **0**? In a strictly typed language such questions don’t arise. But in our language they do and they have answers: the sum of **T** and **0** is **0**, not **T**. An accurate statement of our version of the right identity law is

Theorem.

(IMPLIES (NUMBERP I) (EQUAL (PLUS I 0) I))

or, perhaps more conveniently,

Theorem.

(EQUAL (PLUS I 0) (FIX I)).

Often it is necessary to define new concepts simply to state the desired properties of given ones. For example, suppose we wished to formalize the notion that **SORT** produces ordered output. To express this as a formula we must define the “predicate” that checks whether a list is ordered. One suitable definition is

Definition.

(ORDEREDP L)

=

**(IF (NLISTP L) ; An empty list is ordered.
T**

**(IF (NLISTP (CDR L)) ; A list of length 1 is too.
T**

**(AND (LEQ (CAR L) ; Otherwise, check that
(CADR L)) ; the first two elements
; are in the right relation
; and that the rest of L
; is ordered.**

(ORDEREDP (CDR L))))

Thus, **(ORDEREDP '(1 2 2 5 27 99))** is **T**, but **(ORDEREDP '(1 2 2 5 27 17 99))** is **F**.

The formalization of “**SORT** produces ordered output” is then

Theorem.

(ORDEREDP (SORT L)).

Of course, is **ORDEREDP** really what we mean by “ordered”? That is the hard problem. Furthermore, even if we are convinced that **ORDEREDP** is correctly defined, there is the additional problem: “If a function *does* produce ordered output, do I know it is a correct sort function?” The answer is “no.” For example, the function

Definition.

(FAST-SORT L) = NIL.

has the property that it produces ordered output.

Generally, when we say a function is a “sort function” we mean that it permutes its input into order. This raises the question “How do we formalize the notion that one list is a permutation of another?” We define **PERM** so that it

scans the first list and attempts to find each element as a member of the second list. If some element is not found, the two lists are not permutations of one another. When an element is found in the second list, that occurrence of the element is deleted from the second list so that it is not used to account for duplications in the first list.

Definition.

```
(PERM X Y)
=
(IF (NLISTP X)
    (NLISTP Y)
    (AND (MEMBER (CAR X) Y)
         (PERM (CDR X)
                (DELETE (CAR X) Y))))).
```

(DELETE E L) deletes the first occurrence of E in L:

Definition.

```
(DELETE E L)
=
(IF (NLISTP L)
    L
    (IF (EQUAL E (CAR L))
        (CDR L)
        (CONS (CAR L) (DELETE E (CDR L))))).
```

Again, is (PERM X Y) what we mean by the informal remark that X is a permutation of Y? We can test PERM of course and see that (PERM '(1 2 3 4) '(3 1 2 4)) is T, and (PERM '(1 2 3 4) '(3 1 1 4)) is F. In addition, we can prove theorems about PERM, for instance that it is reflexive and transitive:

Theorem.

```
(PERM X X)
```

Theorem.

```
(IMPLIES (AND (PERM X Y)
               (PERM Y Z))
          (PERM X Z))
```

But in the end, we are left with the fundamental formalization question.

If we accept both ORDEREDP and PERM, then a more complete correctness conjecture for SORT is

Theorem.

```
(AND (PERM L (SORT L))
     (ORDEREDP (SORT L))).
```

3.3. Dealing with Omissions

3.3.1. *Mutual Recursion*

The logic formally prohibits mutual recursion. However, as discussed in the primer, the prohibition can be effectively skirted by defining a single function whose body is a syntactic combination of the mutually recursive definitions. See page 46 of the primer for the details.

3.3.2. *Quantification*

Most users have been exposed to predicate calculus and have some familiarity with the unbounded universal quantifier, \forall (pronounced “for all”), and the unbounded existential quantifier, \exists (pronounced “for some”). In the attempt to formalize an intuitive statement many users therefore go through a stage in which they write down or at least think of quantified versions of the terms in our logic. They then convert these quantified sentences into terms which are interprovable in the two systems. This is not an entirely precise remark because we haven’t formalized a version of our logic containing \forall and \exists , but the idea should be clear.

The most commonly used observation is that if you want to prove $\forall x p(x)$ in predicate calculus, you should prove $(p\ x)$ in our system, where $(p\ x)$ is the translation into our language of the predicate $p(x)$. That is, the axioms, definitions, and theorems of our logic can be thought of as having implicit universal quantification on the far outside. For example

Problem. State that **TIMES** is commutative.

In the imagined extension of the logic to include \forall we wish to prove

$$\forall I \forall J (\text{EQUAL } (\text{TIMES } I\ J) (\text{TIMES } J\ I)).$$

But this is provable in the imagined extension of the logic precisely if

$$(\text{EQUAL } (\text{TIMES } I\ J) (\text{TIMES } J\ I))$$

is provable in the logic provided. As we have seen before, if this term is a theorem, then *for all* values of **I** and **J** the two **TIMES** expressions are equivalent.

This example is deceptively simple because it suggests that universal quantification, at least, can simply be ignored. This is not the case and great care is called for when eliminating universal quantifiers from any position other than

the outside of a formula. We return to this in a moment, but first we discuss the similarly simple case of an outermost existential quantifier.

Problem. State that there exists a right zero for **TIMES**.

By “a right zero for **TIMES**” we mean a number, **z**, such that **(TIMES I z) = 0**, for all **I**. Thus, the existence conjecture is

$$\exists \mathbf{z} \ \forall \mathbf{I} \ (\mathbf{EQUAL} \ (\mathbf{TIMES} \ \mathbf{I} \ \mathbf{z}) \ \mathbf{0}). \quad [1]$$

Since we do not have quantification, we are forced to exhibit a suitable **z**. That is, we drop the $\exists \mathbf{z}$ and replace all free occurrences of **z** by a term that has the alleged property. A suitable term is **0**:

$$\forall \mathbf{I} \ (\mathbf{EQUAL} \ (\mathbf{TIMES} \ \mathbf{I} \ \mathbf{0}) \ \mathbf{0}).$$

If we proved this, the law of \exists -introduction in predicate calculus would immediately let us prove [1]. By the previous observations about universal quantifiers on the outside, we can drop the $\forall \mathbf{I}$ above and are left with

Theorem.

$$(\mathbf{EQUAL} \ (\mathbf{TIMES} \ \mathbf{I} \ \mathbf{0}) \ \mathbf{0})$$

as a formalization of the statement that there exists a right zero for **TIMES**.

Problem. State that **0** is the unique right zero for **TIMES**.

If by this remark we mean “If **z** is a right zero for **TIMES** then **z** is **0**,” the remark is not valid. **0** is not the only right zero for **TIMES**. **T** is also a right zero:

Theorem.

$$(\mathbf{EQUAL} \ (\mathbf{TIMES} \ \mathbf{I} \ \mathbf{T}) \ \mathbf{0}).$$

So by “**0** is unique” here we must mean “**0** is the only number” or “**0** is unique up to coercion by **FIX**.”

That said, a suitable rendering in the imagined extension of the logic is

$$(\mathbf{IMPLIES} \ [\forall \mathbf{I} \ (\mathbf{EQUAL} \ (\mathbf{TIMES} \ \mathbf{I} \ \mathbf{z}) \ \mathbf{0})] \ (\mathbf{EQUAL} \ (\mathbf{FIX} \ \mathbf{z}) \ \mathbf{0})). \quad [2]$$

The hypothesis says “**z** is a right zero,” and the conclusion says “**z** is (coerced to) **0**.”

Very often, new users make the mistake of simply dropping the universal quantifier in the hypothesis, producing, in this case,

$$(\mathbf{IMPLIES} \ (\mathbf{EQUAL} \ (\mathbf{TIMES} \ \mathbf{I} \ \mathbf{z}) \ \mathbf{0}) \ (\mathbf{EQUAL} \ (\mathbf{FIX} \ \mathbf{z}) \ \mathbf{0})). \quad [3]$$

But formula [3], while in our language, is not at all equivalent to [2] and, indeed,

is not a theorem! To see that [3] is not a theorem, evaluate it with **I** equal to 0 and **Z** equal to 47. The hypothesis, (**EQUAL** (**TIMES** 0 47) 0), is true but the conclusion, (**EQUAL** (**FIX** 47) 0), is false. Hence, the implication is false.

What is wrong? The problem is that you simply cannot drop a universal quantifier in the hypothesis. The hypothesis in [2] is very strong. Given a **Z**, the hypothesis

$$[\forall \mathbf{I} \text{ (EQUAL (TIMES } \mathbf{I} \text{ } \mathbf{Z}) 0)]$$

says that (**TIMES** **I** **Z**) is 0 for all **I**. The hypothesis (**EQUAL** (**TIMES** **I** **Z**) 0) in [3] is much weaker. Given an **I** and a **Z** it says (**TIMES** **I** **Z**) is 0—it doesn't say that **Z** is a right zero for **TIMES**.

More generally, in predicate calculus, $((\forall x p) \rightarrow q)$ is not equivalent to $\forall x(p \rightarrow q)$. The tendency simply to ignore universal quantifiers is encouraged by the fact that $(p \rightarrow (\forall x q))$ is equivalent to $\forall x(p \rightarrow q)$, provided x is not free in p , and the fact that it is conventional to have implicit universal quantification of all free variables on the far outside of any theorem.

So how do we formalize [2] within the logic? In the case where the universal quantification is over a finite domain, for example, had the hypothesis read “for all **I** between **N** and **M**, ...,” the standard approach is to replace it with a **FOR** statement or with a call of a recursively defined function that checks the alleged property of **I** for each value in the domain. This approach is not applicable here because we are dealing with an unbounded universal quantifier.

Two common techniques are used, both of which rely upon an understanding of predicate calculus and Skolemization. The first is to use the fact that if x is not free in q , then $((\forall x p) \rightarrow q)$ is equivalent to $\exists x(p \rightarrow q)$. Thus, we can render [2] as

$$\forall \mathbf{Z} \exists \mathbf{I} \text{ (IMPLIES (EQUAL (TIMES } \mathbf{I} \text{ } \mathbf{Z}) 0) \\ \text{(EQUAL (FIX } \mathbf{Z}) 0))}.$$

What does this formula say? It says that for any **Z** you can find an **I** such that if (**TIMES** **I** **Z**) is 0, then **Z** is (coerced to) 0. To render that into our logic we must exhibit a suitable **I** given any **Z**. In particular, we define a “witness function,” **I-WITNESS**, drop the existential quantifier, $\exists \mathbf{I}$, and replace all free **I**s by (**I-WITNESS** **Z**).

$$\forall \mathbf{Z} \text{ (IMPLIES (EQUAL (TIMES (I-WITNESS } \mathbf{Z}) \mathbf{Z}) 0) \\ \text{(EQUAL (FIX } \mathbf{Z}) 0))}.$$

What is a suitable definition of (**I-WITNESS** **Z**)? It must be a number such that if **Z** is a right zero for it, then **Z** is a right zero for everything. Clearly, we could define **I-WITNESS** to be the constant function that returns 1—in this case the witness need not take account of which particular **Z** is being considered.

Eliminating **I-WITNESS** altogether, replacing the call by **1**, and dropping the universal quantifier as previously explained, we get the following formula as a formalization of [2]:

Theorem.

(IMPLIES (EQUAL (TIMES 1 Z) 0)
(EQUAL (FIX Z) 0)).

To many readers the following statement, which is stronger than [2], is a more convincing formalization of the uniqueness of the right zero for **TIMES**:

Theorem.

(IFF (EQUAL (TIMES I J) 0)
(OR (EQUAL (FIX I) 0)
(EQUAL (FIX J) 0))).

This formula says that the product of two numbers is **0** iff one of the two arguments is **0**.

A second approach to formalizing [2] is to Skolemize the formula and appeal to the deduction law of predicate calculus. In particular, [2] is provable in predicate calculus iff

Theorem.

(EQUAL (FIX (Z)) 0)

can be proved assuming the new

Axiom.

(EQUAL (TIMES I (Z)) 0)

where **Z** is a new, undefined function symbol of no arguments. This is an unusual way to formalize the problem because instead of producing a formula to prove, we produce a sequence of logical acts that if admissible in our logic assures us that the desired formula is a theorem of another logic. This approach is commonly used in program verification. See for example our work on the verification of Fortran [26]. For practical reasons we do not recommend this approach to handling universally quantified hypotheses because our theorem prover is not designed to deduce consequences from arbitrary axioms.

Problem. State the unique prime factorization theorem. Traditionally, a natural number p is *prime* iff $p > 1$ and the only divisors of p are 1 and p . A *prime factorization* of a number n is a finite set of pairs $\{ \langle p_1, i_1 \rangle \dots \langle p_k, i_k \rangle \}$ with the property that each p_j is prime, each i_j is greater than 0, p_i is p_j iff i is j , and the product $p_1^{i_1} \dots p_k^{i_k}$ is n . The theorem that every natural number has a unique prime factorization is then: for every natural number n there exists a prime factorization, and if both s_1 and s_2 are prime factorizations of n then $s_1 = s_2$.

Note that the notion of prime, as traditionally defined, involves an unbounded universal quantifier: no number besides 1 and p divides p . Rather than attempt to capture that unbounded quantifier, we adopt the definition of prime that says “no number less than p divides p , except 1.” We can define this concept recursively since the quantification is now over the finite domain of numbers less than p .

Definitions.

```
(PRIME P)
=
(AND (NOT (ZEROP P))
      (NOT (EQUAL P 1))
      (PRIME1 P (SUB1 P)))
```

where

```
(PRIME1 X Y)
=
(IF (ZEROP Y)
    F
    (IF (EQUAL Y 1)
        T
        (AND (NOT (DIVIDES Y X))
              (PRIME1 X (SUB1 Y)))))).
```

This is not, *a priori*, the same concept as *prime* above, for there may be some **PRIMES** that fail to be *prime* because they are divisible by any of the unbounded number of naturals greater than themselves. We accept that **PRIME** is *prime* only after we have proved that no number is divisible by a number greater than itself.

Rather than use sets of pairs to define prime factorizations we use lists of primes.

We say **S** is a **PRIME-FACTORIZATION** of **N** iff **S** is a list of **PRIMES** and the product of the elements of **S** is **N**:

Definition.

```
(PRIME-FACTORIZATION S N)
=
(AND (FOR X IN S ALWAYS (PRIME X))
      (EQUAL N (FOR X IN S MULTIPLY X)))
```

Note that here, too, the quantification is bounded so our **FOR** function is adequate.

Use of the **FOR** quantifier is unnecessary. An alternative approach is to define the functions **PRODUCT** and **PRIME-LIST** so that they recursively

determine what the product is and whether **L** is a list of primes.

Definitions.

```
(PRIME-LIST L)
=
(IF (NLISTP L)
    T
    (AND (PRIME (CAR L))
          (PRIME-LIST (CDR L)))))

(PRODUCT L)
=
(IF (NLISTP L)
    1
    (TIMES (CAR L)
            (PRODUCT (CDR L)))).
```

By defining factorizations to be lists of primes we avoid the use of sets, but this returns to haunt us when we must state uniqueness. Our statement of uniqueness is that two lists of primes with the same product are equal up to order, which is to say, they are permutations of each other. We state the uniqueness part of the main theorem as follows:

Theorem.

```
(IMPLIES (AND (PRIME-FACTORIZATION S1 N)
               (PRIME-FACTORIZATION S2 N))
          (PERM S1 S2)).
```

Finally, we must state the existence part of the theorem without using an existential quantifier. This we do by defining a “witness function,” **PRIME-FACTORS**, that takes any given positive number **N** and returns an alleged factorization. The existence theorem is then

Theorem.

```
(IMPLIES (AND (NUMBERP N)
               (NOT (EQUAL N 0)))
          (PRIME-FACTORIZATION (PRIME-FACTORS N) N)).
```

That is, we not only prove there exists a factorization, we exhibit a constructive function for obtaining one.

The definition we use for **PRIME-FACTORS** is irrelevant to the intended understanding of the existence theorem. In fact, the definition is relatively complicated: we define the notion of the greatest factor of a number and then we define the prime factors as follows. The prime factors of 0 and 1 is the empty list; the prime factors of a prime is the singleton list containing the prime; and the prime factors of any other number **n** is obtained by appending the prime

factors of the greatest factor, **m**, of **n**, to the prime factors of the quotient of **n** by **m**. That this is an admissible definition follows from the observations that both **m** and the quotient of **n** by **m** are smaller than **n**. The reason we defined **PRIME-FACTORS** this way was to make the proof of the existence theorem easier. It is trivial to show that (**PRIME-FACTORS N**) yields a list of primes. It is also relatively straightforward to show that the product of the prime factors is **N**, given that the greatest factor of **N** divides **N** if **N** is not prime.

In summary, we use recursion to capture the standard quantifiers, proving bounds on the universal quantification when necessary. Of course, **FOR** can be used in straightforward cases. We use lists where sets are traditionally used. Within this setting we have proved such deep results as Fermat's theorem [27], Wilson's theorem [120], and Gauss's law of quadratic reciprocity [121]. Such standard mathematical techniques as modulo arithmetic, the pigeon hole principle, and one-to-one mappings have been formalized, proved, and successfully used in other proofs.

3.3.3. *Infinites*

Problem. State the theorem that there are an infinite number of primes.

One formalization of this idea is that for every prime **P** there exists a prime bigger than **P**—and of course, there *is* at least one prime, witness **7**. To eliminate the quantifier define the function **BIGGER-PRIME** which takes a number **N** and returns a prime bigger than **N**. Then prove

Theorem.

```
(IMPLIES (PRIME P)
  (AND (PRIME (BIGGER-PRIME P))
    (LESSP P (BIGGER-PRIME P))))
```

An appropriate **BIGGER-PRIME** can be defined by searching through the numbers from **P+1** to **P!-1**. That at least one of those is prime follows from the fact that no prime less than or equal to **P** divides **P!-1**. Thus, if **P!-1** is itself not prime, its greatest prime factor is greater than **P**.

If the above theorem is not enough, it can be used as the main stepping stone to another: To each natural number there corresponds a unique prime. Again, the existential quantifier is eliminated by a constructive function, **NTH-PRIME**, with the following two properties:

Theorem.

```
(IMPLIES (NUMBERP N)
  (PRIME (NTH-PRIME N)))
```

and

Theorem.

```
(IMPLIES (AND (NUMBERP N)
              (NUMBERP M)
              (EQUAL (NTH-PRIME N)
                    (NTH-PRIME M)))
         (EQUAL N M))
```

Another kind of infinity with which we frequently deal has to do with processes that “run forever.”

Problem. Formalize a “message flow modulator,” a device that sits on a communications line and filters the incoming messages into two streams according to some predicate.

Some formalists in computer science are very fond of using infinite sequences to represent the three message streams. However, for very many purposes finite sequences suffice. For example, we might formalize the modulator as a function **MFM** that takes a single message **msg** as its argument and returns a pair **<tag, msg>** as its result, where **tag** indicates to which output stream the outgoing message is assigned. The system is then formalized as

Definition.

```
(SYSTEM S)
=
(IF (NLISTP S)
    NIL
    (CONS (MFM (CAR S))
          (SYSTEM (CDR S)))).
```

Here we formalize the system as a function which takes an incoming stream of messages, **S**, and produces a list of pairs, each of which is of the form **<tag, msg>** indicating a destination and the associated input message.⁴

We might then prove the theorem that the messages sent to tag **I** are a subset of the incoming messages:

```
(SUBSETP (FOR PAIR IN (SYSTEM S)
              WHEN (EQUAL I (CAR PAIR))
              COLLECT (CDR PAIR))
         S)
```

Many people object to this formalization of the system because it suggests that

⁴The function **SYSTEM** could have been defined equivalently as **(FOR MSG IN S COLLECT (MFM MSG))**. However, the recursive definition more easily accommodates the complexities of more realistic models.

only a finite number of messages come down the communications line. We have two views on the matter, depending on how ornery we're feeling. The first is that, indeed, no communications line yet has had an infinite number of messages on it and no communications line is ever going to. The second view is that the variable **S** in the model is not to be thought of as the entire history of the communications line but merely some initial segment of it, and the answer returned by (**SYSTEM S**) is not the entire output of the system but only the output in response to that initial segment. The theorem means that the messages output to tag **I** *so far* are a subset of the incoming messages *so far*.

3.3.4. Higher Order Logic

Our logic is first order. However, the ability to constrain and to use functional instantiation often allows apparently higher order acts. In this section we illustrate the use of these features of the logic to overcome the omission of higher order functions and operations. Some of the same omissions can be addressed with our **V&C\$** and **FOR** functions, discussed later in this chapter.

Problem. Call a function symbol, **fn**, of two arguments “doubly commutative” iff **fn** is commutative and (**fn x (fn y z)**) is (**fn y (fn x z)**). Prove that every doubly commutative function symbol is associative.

In a suitable higher order logic with full first order quantification, one might make definitions such as:

Definition.

(DOUBLY-COMMUTATIVE *fn*)
 =
 (AND $\forall X \forall Y$ (EQUAL (*fn* X Y) (*fn* Y X))
 $\forall X \forall Y \forall Z$ (EQUAL (*fn* X (*fn* Y Z))
 (*fn* Y (*fn* X Z))))),

and

Definition.

(ASSOCIATIVE *fn*)
 =
 $\forall X \forall Y \forall Z$ (EQUAL (*fn* (*fn* X Y) Z)
 (*fn* X (*fn* Y Z)))

and then prove

Theorem

(IMPLIES (DOUBLY-COMMUTATIVE *fn*)
 (ASSOCIATIVE *fn*)).

We cannot make such definitions because they take functions as arguments (not to mention that they involve unbounded quantification over the possible arguments to the functions). However, with introduction by constraint and functional instantiation we can deal with some aspects of the problem.

Let us introduce the function symbol **FN** of two arguments as follows.

Constraint. DOUBLY-COMMUTATIVE

Constrain **FN** to have the property

$$\begin{aligned} &(\text{AND } (\text{EQUAL } (\text{FN } X \ Y) \ (\text{FN } Y \ X)) \\ &\quad (\text{EQUAL } (\text{FN } X \ (\text{FN } Y \ Z)) \\ &\quad \quad (\text{FN } Y \ (\text{FN } X \ Z))))). \end{aligned}$$

A suitable witness for this constraint is the function **PLUS**. Once introduced, we may think of **FN** as any doubly commutative function symbol. We can then undertake to prove that **FN** is associative:

Theorem. FN-IS-ASSOCIATIVE

$$(\text{EQUAL } (\text{FN } (\text{FN } X \ Y) \ Z) \ (\text{FN } X \ (\text{FN } Y \ Z)))).$$

Indeed, if the constraining axiom for **FN** shown above is of rule class **REWRITE**, the proof of associativity is fully automatic. In particular, the sequence of rewrites performed is shown below, where each replacement is justified by one of the two equalities in the **DOUBLY-COMMUTATIVE** constraint:

$$\begin{aligned} (\text{FN } (\text{FN } X \ Y) \ Z) &= (\text{FN } Z \ (\text{FN } X \ Y)) \\ &= (\text{FN } X \ (\text{FN } Z \ Y)) \\ &= (\text{FN } X \ (\text{FN } Y \ Z)). \end{aligned}$$

It is clear that this exercise captures one part of the problem, namely, showing why “double commutativity” implies associativity. However, is this result useful?

Problem. Suppose **CONVOLVE** is a function that has been proved to be doubly commutative. Prove that it is associative.

In the imagined higher-order logic used above, the supposition is just that **(DOUBLY-COMMUTATIVE 'CONVOLVE)** is a theorem. The desired conclusion is that **(ASSOCIATIVE 'CONVOLVE)** is a theorem. The conclusion can be obtained from the supposition by *modus ponens*.

In our case, functional instantiation is used instead. In particular, the associativity of **CONVOLVE** can be deduced from **FN-IS-ASSOCIATIVE** by functional instantiation, replacing **FN** by **CONVOLVE**. Functional instantiation requires that **CONVOLVE** satisfy the axioms on **FN**. In this case, there is just one, the constraint **DOUBLY-COMMUTATIVE**. Hence, at the cost of proving (or noting the previous proof of) the double commutativity of **CONVOLVE** we conclude that it is associative.

Problem. Show that if a function is doubly commutative then mapping it over a list produces the same result as mapping it over the reverse of the list.

To formalize this problem we must first agree on the sense of “mapping” a function over a list. Suppose \mathbf{x} is a list whose elements are $\mathbf{x}_1, \dots, \mathbf{x}_n$. Then $(\mathbf{map} \ fn \ \mathbf{x} \ \mathbf{b})$ is $(fn \ \mathbf{x}_1 \ (fn \ \mathbf{x}_2 \ \dots (fn \ \mathbf{x}_n \ \mathbf{b}) \dots))$. This operation is called “foldr” in the higher-order functional language SASL [138]. Because the desired **map** function takes a function as an argument, we cannot define it.

We can define the notion of mapping a particular function over a list. For example, the following function maps **PLUS** over a list.

Definition.

```
(MAP-PLUS X B)
=
(IF (NLISTP X)
  B
  (PLUS (CAR X)
    (MAP-PLUS (CDR X) B)))).
```

Thus, $(\mathbf{MAP-PLUS} \ \mathbf{x} \ 0)$ is the sum of the elements of \mathbf{x} .

Since we are interested in the results of mapping with doubly commutative functions, and we have constrained **FN** to be any such function, we might therefore consider

Definition.

```
(MAP-FN X B)
=
(IF (NLISTP X)
  B
  (FN (CAR X)
    (MAP-FN (CDR X) B)))).
```

We can think of **MAP-FN** as the most general mapping function for doubly commutative functions in the sense that if **MAP-fn** is a function that maps with some particular doubly commutative function, then **MAP-fn** enjoys any property **MAP-FN** does. We illustrate this in a moment.

Having defined **MAP-FN** we might formalize the challenge problem above as

Theorem. **MAP-FN-REVERSE**

```
(EQUAL (MAP-FN (REVERSE X) B)
  (MAP-FN X B)).
```

This formula says that **MAP-FN** returns the same thing on $(\mathbf{REVERSE} \ \mathbf{x})$ as on \mathbf{x} . We here are imagining that **REVERSE** is defined to be the function that reverses the order of the elements in a list. The above theorem is easy to prove, at least for the obvious definition of **REVERSE**.

Again, the utility question arises: can we use **MAP-FN-REVERSE** to derive the analogous property about, say, **MAP-PLUS**? The answer, again, is “yes, with functional instantiation.” In particular,

Theorem. MAP-PLUS-REVERSE
(EQUAL (MAP-PLUS (REVERSE X) B)
(MAP-PLUS X B))

can be derived from **MAP-FN-REVERSE** by functional instantiation, replacing **MAP-FN** by **MAP-PLUS** and replacing **FN** by **PLUS**. Functional instantiation requires that the new functions satisfy the axioms about the old functions. In particular, every axiom about the old functions must yield a theorem when the old function symbols are replaced by the new ones.

The only axiom about **MAP-FN** is its definition. When the old symbols are replaced by the new ones, we obtain

Theorem?
(MAP-PLUS X B)
 =
(IF (NLISTP X)
B
(PLUS (CAR X)
(MAP-PLUS (CDR X) B))) .

This formula is a theorem because of the definition of **MAP-PLUS**.

There are now two axioms about **FN**. The first is the **DOUBLY-COMMUTATIVE** constraint. When the old symbols are replaced by the new ones we obtain

Theorem?
(AND (EQUAL (PLUS X Y) (PLUS Y X))
(EQUAL (PLUS X (PLUS Y Z))
(PLUS Y (PLUS X Z)))) .

That is, we are obliged to prove that **PLUS** has the double commutativity property. The second axiom about **FN** is the definition of **MAP-FN**, which, when functionally instantiated, yields the **MAP-PLUS** theorem (definition) just shown.

To summarize, we can use our general result about **MAP-FN** to derive the analogous result about some particular mapping function, provided the function mapped is doubly commutative.

3.4. Dealing with Features

In this section we discuss two rather peculiar features of the logic: the interpreter, **V&C\$**, and the bounded quantifier, **FOR**. The reader should see [30] for a deeper discussion of these functions. Much of this section is extracted or paraphrased from that paper.

3.4.1. Termination Theorems with *V&C\$*

Problem. Prove the termination properties of the partial function described by

```
(APP X Y)
  ⇐
(IF (EQUAL X (QUOTE NIL))
  Y
  (CONS (CAR X) (APP (CDR X) Y))).
```

The recurrence described here is similar to **APPEND**'s but the termination condition is that **X** is **NIL** rather than that **X** is not a **LISTP**. Note that if the first argument of **APP** is **0**, the computation of **APP** runs forever:

```
(APP 0 Y)
  ⇐
(CONS 0 (APP 0 Y))
  ⇐
(CONS 0 (CONS 0 (APP 0 Y)))
  ⇐
(CONS 0 (CONS 0 (CONS 0 (APP 0 Y))))
  ⇐
...
```

And in general, since the **CDR** of a non-**LISTP** is **0**, if the first argument to **APP** does not end in **NIL** the computation runs forever. That is the first “termination property” of **APP**. The second is that when it terminates, **APP** returns the same thing as **APPEND**. We wish to formalize these two statements.

An easy subproblem is to formalize what we mean by “ends in **NIL**.” We define **PROPERP** to recognize those lists that end in **NIL**:

Definition.

```
(PROPERP L)
  =
(IF (NLISTP L) (EQUAL L NIL) (PROPERP (CDR L))).
```

That is, a non-**LISTP** is **PROPERP** only if it is **NIL**, and a **LISTP** is **PROPERP** precisely when its **CDR** is.

Now how do we get a handle on termination?

Consider for a moment the proposed definition similar to the description of **APP** above:

Definition?

```
(APP X Y)
=
(IF (EQUAL X NIL)
    Y
    (CONS (CAR X)
          (APP (CDR X) Y))).
```

This equation is inadmissible under the principle of definition because there is no measure of the arguments that decreases in a well-founded sense. Were the above equation an axiom we could derive

$$(\text{APP } 0 \ 1) = (\text{CONS } 0 \ (\text{APP } 0 \ 1)),$$

contradicting the theorem $Y \neq (\text{CONS } X \ Y)$.

To investigate the termination properties of **APP** we cannot add the equation “defining” **APP** as an axiom! We use **V&C\$** instead.

First, we define **APP** using **EVAL\$** as follows:

Definition.

```
(APP X Y)
=
(EVAL$ T '(IF (EQUAL X (QUOTE NIL))
              Y
              (CONS (CAR X)
                    (APP (CDR X) Y))))
(LIST (CONS 'X X)
      (CONS 'Y Y)).
```

As noted in the primer (page 59) and explained in detail in Chapter 4 (page 176), one effect of this definition is to add as axioms the three following equations:

Axioms.

```
(SUBRP 'APP) = F
```

```
(FORMALS 'APP) = '(X Y)
```



```

(BODY 'APP)
=
'(IF (EQUAL X (QUOTE NIL))
  Y
  (CONS (CAR X) (APP (CDR X) Y))))

```

Now recall that **V&C\$** takes (the quotation of) a term and an association list assigning values to (the quotations of) variable symbols. **V&C\$** returns either a pair, **<v, c>**, indicating the value and cost of the evaluation of the term, or else returns **F** if no cost is sufficient. Thus, the intuitive idea that an expression “runs forever” is here formalized by the assertion that the **V&C\$** of the expression is **F**.

Here then is the statement of the first termination property of **APP** that **(APP X Y)** terminates if and only if **X** is proper:

Theorem.

```

(IFF (V&C$ T '(APP X Y)
      (LIST (CONS 'X X)
            (CONS 'Y Y)))
      (PROPERP X))

```

Note that we had to quote the expression, **(APP X Y)** and supply the map between the quoted variables **'X** and **'Y** and the variables **X** and **Y**.

The second termination property of **APP** is that if **(APP X Y)** terminates its value is **(APPEND X Y)**. The hypothesis that the expression terminates is just that the **V&C\$** of it is non-**F**. The delivered value is the **CAR** of the pair returned by **V&C\$**. Thus, here is an appropriate formalization:

Theorem.

```

(IMPLIES (V&C$ T '(APP X Y)
          (LIST (CONS 'X X)
                (CONS 'Y Y)))
         (EQUAL (CAR (V&C$ T '(APP X Y)
                              (LIST (CONS 'X X)
                                      (CONS 'Y Y)))))
          (APPEND X Y))).

```

Problem. State that (**RUSSELL**) does not terminate, where

$$(\text{RUSSELL}) \Leftarrow (\text{NOT } (\text{RUSSELL})).$$

The solution is exactly analogous to that above. Define **RUSSELL** as

Definition.

```
(RUSSELL)
=
(EVAL$ T '(NOT (RUSSELL)) NIL)
```

so as to add the appropriate axioms for **FORMALS** and **BODY**. Then the statement that (**RUSSELL**) doesn't terminate is

Theorem.

```
(EQUAL (V&C$ T '(RUSSELL) NIL)
F).
```

Problem. Formalize the termination properties of the 91-function, where

```
(F91 X)
  <=
(IF (LESSP 100 X)
    (DIFFERENCE X 10)
    (F91 (F91 (PLUS X 11)))).
```

It turns out that this function is total and computes (**G91 X**), where

Definition.

```
(G91 X)
=
(IF (LESSP 100 X)
    (DIFFERENCE X 10)
    91).
```

To formalize this, we define **F91** as

Definition.

```
(F91 X)
=
(EVAL$ T '(IF (LESSP '100 X)
              (DIFFERENCE X '10)
              (F91 (F91 (PLUS X '11)))))
  (LIST (CONS 'X X)).
```

Then

Theorem.

```
(AND (V&C$ T '(F91 X) (LIST (CONS 'X X)))
      (EQUAL
        (CAR (V&C$ T '(F91 X) (LIST (CONS 'X X))))
        (G91 X))).
```

The first conjunct states that **F91** is total. The second states that its value is that of **G91**.

In [30] we show how such theorems can be proved within the logic. Leading the theorem prover to such proofs is often difficult. We have done it for the theorems mentioned here but do not yet have enough experience to pass on reliable advice. The interested reader should see example file 14.

3.4.2. Schematic Theorems about FOR

Suppose the user of the logic desires to discuss the list obtained by doubling the elements of **L**. One way to proceed is to define (**DOUBLE-LIST L**) as follows:

Definition.

```
(DOUBLE-LIST L)
=
(IF (NLISTP L)
    NIL
    (CONS (TIMES 2 (CAR L))
          (DOUBLE-LIST (CDR L)))).
```

For example, (**DOUBLE-LIST '(1 2 3 4)**) = **'(2 4 6 8)**.

A useful theorem about **DOUBLE-LIST** is that it “distributes” over **APPEND**:

Theorem.

```
(DOUBLE-LIST (APPEND A B))
=
(APPEND (DOUBLE-LIST A)
        (DOUBLE-LIST B)).
```

However, suppose that in addition, the user wished to refer to the list obtained by adding 1 to every element of **L**. In the spirit that led to the definition of **DOUBLE-LIST** the user would have to define the function **ADD1-LIST**, so that (**ADD1-LIST '(1 2 3 4)**) = **'(2 3 4 5)**. Should it be necessary to use the fact that **ADD1-LIST** distributes over **APPEND**, that fact would

have to be proved. One is tempted to say “proved again.” But since **DOUBLE-LIST** and **ADD1-LIST** are different function symbols, the first lemma cannot be used to shorten the proof of the second.

An alternative is to use the quantifier **FOR** and write such expressions as

(FOR X IN L COLLECT (TIMES 2 X))

and

(FOR X IN L COLLECT (ADD1 X)).

The first expression is equivalent to (**DOUBLE-LIST** L) and the second is equivalent to (**ADD1-LIST** L).

It is possible to state very general lemmas about quantifiers. Consider for example the schematic form

$$\begin{aligned} &(\text{FOR } V \text{ IN } (\text{APPEND } A \text{ } B) \text{ COLLECT } body(V)) \\ &= \\ &(\text{APPEND } (\text{FOR } V \text{ IN } A \text{ COLLECT } body(V)) \\ &\quad (\text{FOR } V \text{ IN } B \text{ COLLECT } body(V))), \end{aligned}$$

where *body* is understood here to be a second order variable. (Our formalization does not introduce such variables or other new syntactic classes. We adopt this notation now only for expository purposes in the same spirit in which we earlier used \forall and \exists in our formulas.) This lemma is easy to prove because no time is wasted considering special properties of the particular *body* used. This lemma is more useful than the fact that **DOUBLE-LIST** distributes over **APPEND** since it also handles the analogous property for **ADD1-LIST** and other such functions.

The introduction of quantifiers has three major attractions. First, quantifiers are conceptually clarifying. Logical relationships that would otherwise be buried inside of recursive definitions are lifted out into the open. The users and readers of the logic need no longer invent and remember names for many “artificial” recursive functions. Second, it is possible to state general purpose, schematic theorems about quantifiers, thus reducing the number of theorems needed. Third, these schematic theorems are generally easier to prove than the corresponding theorems about recursively defined instances of the quantifiers, because irrelevant concepts have been abstracted out of the theorems. Thus, quantifiers are a boon to the reader, to the writer, and to the theorem prover. However, you have to know how to use them and that is what this section is about.

Problem. State that summation distributes over **APPEND**. That is, formalize the following schematic theorem:

$$\begin{aligned} &(\text{FOR } V \text{ IN } (\text{APPEND } A \ B) \ \text{SUM } \text{body}(V)) \\ &= \\ &(\text{PLUS } (\text{FOR } V \text{ IN } A \ \text{SUM } \text{body}(V)) \\ &\quad (\text{FOR } V \text{ IN } B \ \text{SUM } \text{body}(V))), \end{aligned}$$

Recall the abbreviation conventions for **FOR**. In particular,

```
(FOR X IN (APPEND U V)
  WHEN (MEMBER X B)
  SUM (TIMES X C))
```

is formally represented in the logic and in the implementation by

```
(FOR 'X (APPEND U V)
  '(MEMBER X B)
  'SUM '(TIMES X C)
  (LIST (CONS 'B B) (CONS 'C C))).
```

Note that in the abbreviated form of **FOR** the bound variable symbol **X**, the conditional expression **(MEMBER X B)**, the operator **sum**, and the body **(TIMES X C)**, are written as terms but become quoted constants in the formal representation.

To state a general purpose “schematic” theorem it is necessary to write down a **FOR** expression in which the bound variable, condition, operation, and/or body are all arbitrary terms satisfying given conditions, rather than constants. Thus, schematic theorems about **FOR** use the unabbreviated, formal form of **FOR**, namely **(FOR var range cond op body alist)**.

The schematic theorem expressing the idea that summation distributes over **APPEND** is as follows:

Theorem.

```
(EQUAL (FOR V (APPEND A B) COND 'SUM BODY ALIST)
  (PLUS (FOR V A COND 'SUM BODY ALIST)
    (FOR V B COND 'SUM BODY ALIST))).
```

Note that the equation holds no matter what the bound variable symbol is and no matter what the conditional and body expressions are. For example, the following concrete use of **FOR**, written in the abbreviated form

```
(FOR I IN (APPEND L1 L2) SUM (TIMES 2 I))
```

is just an abbreviation for⁵

⁵The use of **'(TRUE)** in the abbreviated term is actually incorrect. The term abbreviated contains the expression **'(QUOTE *1*TRUE)** instead of **'(TRUE)**. These two have the same value under **EVAL\$**. See page 158.

```
(FOR 'I (APPEND L1 L2) '(TRUE)
'SUM '(TIMES 2 I) NIL).
```

We can instantiate the “schematic” theorem above, replacing **V**—the bound variable—by **'I**, **A** and **B** by **L1** and **L2** respectively, **COND** by **'(TRUE)**, **BODY** by **'(TIMES 2 I)**, and **ALIST** by **NIL**, and derive immediately the equivalence of the concrete **FOR** with

```
(PLUS (FOR 'I L1 '(TRUE) 'SUM '(TIMES 2 I) NIL)
(FOR 'I L2 '(TRUE) 'SUM '(TIMES 2 I) NIL)),
```

which, in abbreviated form is

```
(PLUS (FOR I IN L1 SUM (TIMES 2 I))
(FOR I IN L2 SUM (TIMES 2 I))),
```

as one would have expected.

Problem. Formalize the statement that summation of a **PLUS** expression is the **PLUS** of the summations. Or, in everyday mathematics,

$$\sum_{i=0}^n g(i) + h(i) = \sum_{i=0}^n g(i) + \sum_{i=0}^n h(i);$$

If we had higher order variables g and h we could write

```
(FOR I IN L SUM (PLUS g(I) h(I)))
=
(PLUS (FOR I IN L SUM g(I))
(FOR I IN L SUM h(I))).
```

However, in our logic we don't have such variables, but we don't need them here. It is sufficient simply to use the unabbreviated form of **FOR** and use ordinary variable symbols **G** and **H** to describe (the quotation of) an arbitrary **PLUS** expression:

Theorem.

```
(EQUAL (FOR V L COND 'SUM (LIST 'PLUS G H) ALIST)
(PLUS (FOR V L COND 'SUM G ALIST)
(FOR V L COND 'SUM H ALIST))).
```

Again, note that this theorem can be instantiated so as to let us distribute the summation over the two branches of the body below:

```
(FOR I IN L SUM (PLUS I (SQ I)))
```

It is important to note that the body of the above **FOR** is actually `'(PLUS I (SQ I))` or, put another way, `(LIST 'PLUS 'I '(SQ I))`.⁶

In stating some classic quantifier manipulation theorems it is necessary to put elaborate syntactic constraints on the form of the body expression. Here is a simple example:

Problem. Formalize the theorem that the summation of an expression not involving the bound variable is a constant. In particular, we would like the formal version of the theorem that permits us to reduce $\sum_{i \in s} n$ to simply $|s|*n$.

To do this we must define the notion that the body does not “involve” the bound variable. Suppose we have defined `(FREE-VARS T X)` to be the variables in the (quotation of a) term **X**. Then the first cut at a formalization of the goal is

```
(IMPLIES (NOT (MEMBER V (FREE-VARS T N)))
  (EQUAL (FOR V S '(TRUE) 'SUM N ALIST)
    (TIMES (LENGTH S) N)))
```

However, this is incorrect because what is being summed by the **FOR** is the **EVAL\$** of **N** under successive extensions to **ALIST** produced by binding **V**. Thus, instead of **N** in `(TIMES (LENGTH S) N)` we should write `(EVAL$ T N ALIST)`. An accurate statement of the rule is then

Theorem.

```
(IMPLIES (NOT (MEMBER V (FREE-VARS T N)))
  (EQUAL (FOR V S '(TRUE) 'SUM N ALIST)
    (TIMES (LENGTH S)
      (EVAL$ T N ALIST))))
```

Note that this theorem can be used to simplify a concrete **FOR** provided the conditional expression in the **FOR** is `'(TRUE)` and the bound variable does not occur in the body. The restriction on the conditional expression can be relaxed by replacing **LENGTH** with the quantifier statement that counts the number of times the conditional is true in **S**.

We conclude our discussion of **FOR** by exhibiting the formalization of the concept of **FREE-VARS**. This is particularly appropriate because it forces us to consider the form of the quotations of terms, a topic with which the user must be familiar if accurate schematic theorems are to be written.

⁶One extremely useful application of backquote notation is in connection with schematic theorems about **FOR**. The left-hand side of the theorem above can be written `(FOR V L COND 'SUM '(PLUS ,G ,H) ALIST)` provided the **NQTHM** setting of **BACKQUOTE-SETTING** is in use (see page 301).

The definition of **FREE-VARS** recursively explores terms in the same way that **EVAL\$** does. When **X** is treated as a term there are four cases: **X** is a variable symbol (i.e., (**LITATOM X**)), **X** is a **QUOTED** constant, (i.e., (**LISTP X**) and the **CAR** of **X** is **'QUOTE**), **X** is a function application (i.e., (**LISTP X**) but the **CAR** is not **'QUOTE**), and **X** is of any other form (e.g., a **NUMBERP**). In the case that **X** is a function application we must recursively collect the variables in all of the arguments, and so we are dealing with another case of mutual recursion. Here is a suitable definition:

Definition.

```
(FREE-VARS FLG X)
=
(IF (EQUAL FLG 'LIST)
  (IF (NLISTP X)
    NIL
    (UNION (FREE-VARS T (CAR X))
            (FREE-VARS 'LIST (CDR X)))))

(IF (LITATOM X)
  (LIST X)
(IF (NLISTP X)
  NIL
  (IF (EQUAL (CAR X) 'QUOTE)
    NIL
    (FREE-VARS 'LIST (CDR X))))))
```

Perhaps the most interesting thing about this definition is that it is not necessary for **FREE-VARS** to “know about” **FOR**. The reason is that **FOR** is just like every other function in our logic. It does not really introduce a “bound” variable or have terms as its arguments. Is **I** among the free variables of (**FOR I IN L SUM (PLUS I X)**)? That is to say, is **'I** a member of the list returned by **FREE-VARS** on the quotation of the **FOR** term? The answer is no. The formal term abbreviated by (**FOR I IN L SUM (PLUS I X)**) is (**FOR 'I L '(TRUE) 'SUM '(PLUS I X) (LIST (CONS 'X X))**). The quotation of this term is **'(FOR 'I L '(TRUE) 'SUM '(PLUS I X) (LIST (CONS 'X X)))**. The result of **FREE-VARS** on this is **'(L X)**.

In [30] we show many schematic theorems about **FOR**. In example file 14 we show the events leading to the proofs of those theorems. The reader interested in using **FOR** is urged to look at those events.

3.5. Nondeterminism

Problem. Define a nondeterministic merge function which merges two input streams (finite lists) into a single output stream.

Below is an alleged definition of the concept

Definition.

```
(MERGE A B)
=
(IF (NLISTP A)
    B
    (IF (NLISTP B)
        A
        (IF (RANDOM-BOOL)
            (CONS (CAR A)
                  (MERGE (CDR A) B))
            (CONS (CAR B)
                  (MERGE A (CDR B)))))))
```

where **RANDOM-BOOL** is an undefined function.

The trouble with this definition is that (**RANDOM-BOOL**) is a constant. It is either **F** or non-**F**. Indeed,

Theorem.

```
(IMPLIES (AND (LISTP A)
              (LISTP B))
         (OR (EQUAL (MERGE A B) (APPEND A B))
             (EQUAL (MERGE A B) (APPEND B A)))).
```

Is this a property of a nondeterministic merge function? Probably not.

A typical patch is to make **RANDOM-BOOL** a function of **A** and **B**. Thus, at each stage in the recursion, an element of **A** or an element of **B** is chosen as an undefined function of the current values of **A** and **B**. This function can produce more varied output than the previous one, depending on the properties of **RANDOM-BOOL**. But the mere fact that the element chosen at each stage is a function of the current inputs permits some deductions about the function's behavior. For example, there is no definition of **RANDOM-BOOL** that makes (**MERGE ' (A) ' (B)**) = '**(A B)**' and (**MERGE ' (C A) ' (B)**) = '**(C B A)**', even though both desired outputs are legal merges of the inputs. The reason is that once (**MERGE ' (C A) ' (B)**) chooses to output **C** and recurses to (**MERGE ' (A) ' (B)**), the recursive call must yield '**(A B)**'.

Clearly, **MERGE** can be patched again so that more "state" is given to **RANDOM-BOOL** to prevent this too. Indeed, the following definition of **MERGE** has the desired property:

Definition.

```
(MERGE A B)
=
(MERGE1 A B (ORACLE A B))
```

where

Definition.

```
(MERGE1 A B ORACLE)
=
(IF (NLISTP A)
  B
  (IF (NLISTP B)
    A
    (IF (CAR ORACLE)
      (CONS (CAR A)
        (MERGE1 (CDR A) B (CDR ORACLE)))
      (CONS (CAR B)
        (MERGE1 A (CDR B) (CDR ORACLE)))))))
```

and **ORACLE** is an undefined function.

The basic idea is to provide the nondeterminacy by providing an additional input, the variable **ORACLE** in **MERGE1**, that is entirely responsible for the “nondeterministic choices” and to leave the value of that input unconstrained.

It is the case that for any set of triples $\langle \mathbf{a}, \mathbf{b}, \mathbf{x} \rangle$ with the property that \mathbf{x} is an interleaving of \mathbf{a} and \mathbf{b} , there exists a definition of the function **ORACLE** sufficient to make $(\mathbf{MERGE} \ \mathbf{a} \ \mathbf{b}) = \mathbf{x}$ for every triple in the set. This can be proved within the logic for all finite sets of triples. (Quick! How do you capture the notion “there exists a definition of **ORACLE**?”)

For a more elaborate example of the use of oracles, see the hardware verification work of [73, 70]. There, it is necessary to model the response of memory to read/write requests. Oracles are used to permit arbitrary delays.

An alternative approach to the merge problem is not to *define* **MERGE** but to introduce it by constraint. In particular,

Constraint.

Constrain the function **MERGE** to have the property
 $(\mathbf{PERM} \ (\mathbf{MERGE} \ \mathbf{X} \ \mathbf{Y}) \ (\mathbf{APPEND} \ \mathbf{X} \ \mathbf{Y}))$.

This introduces **MERGE** as a function symbol of two arguments and adds the axiom that it produces a permutation of the concatenation of its arguments. For this event to be admissible the user must supply a “witness” for **MERGE** that satisfies the formula to be added as an axiom. A suitable witness is easy: **MERGE** could be **APPEND**.

Perhaps the biggest philosophical obstacle in the attempt to introduce **MERGE** as a function is that it must be a function. In particular, **(MERGE A B)** is always equal to **(MERGE A B)**, whether **MERGE** is defined or constrained.⁷ A related remark can be made that some people may find reassuring: **(MERGE1 A B ORACLE1)** need not be the same as **(MERGE1 A B ORACLE2)**. Despite the obviousness of these observations we have seen informed users of the logic fail to make similar ones.

For example, suppose you are writing an interpreter within the logic for a programming language like Lisp (see the next section). Suppose your interpreter has the notion of an “environment” by which some program variables are assigned values. Let **(LOOKUP SYM ENV)** be the expression you use to denote the value of the program variable **SYM** in the environment. How do you define **LOOKUP** to cope with the possibility that a program variable is “unbound,” i.e., not assigned a value in the environment? One possibility is to declare an undefined function symbol, **UNDEF**, and to return **(UNDEF)** as the value of such a **LOOKUP**. That captures some of the desired features: it will be impossible to prove much about the type or value of unbound variables. But one consequence of this approach is that all unbound program variables are thus specified to have the same (otherwise undetermined) value. That is, **(UNDEF)** equals **(UNDEF)**.

There are three conventional ways out: (a) change the problem so that non-deterministic behavior is eliminated, e.g., in the programming language example, adopt the specification that all variables have the initial value 0; (b) model the nondeterminism in a trivial, deterministic way but avoid asking questions that depend upon your trivial convention, e.g., define the notion of a well-formed program (in which unbound variables never arise) and include among the hypotheses of all your theorems the constraint that the programs are well-formed; or (c) model the nondeterminism carefully and accurately, e.g., include an oracle in the environment.

3.6. Computing Machines and Programming Languages

Problem. Formalize a simple programming language and prove some program correct.

We discuss this formalization problem at length here because computing machines and programming languages are among the most commonly formal-

⁷Isn't that a silly thing to have to say?

ized systems in our logic. An operational approach is most often taken, but we discuss some other alternatives later. In this example we merely sketch a formalization that is done completely in example file 16.

3.6.1. An Operational Semantics

The programming language we will formalize is a simple, execute-only assembly-level language. It is a simplification of the Piton language formalized in [106]. See example file 31. A typical program in this language is shown below, in an imagined “external” syntax. The program multiplies the contents of register 0 by the contents of register 1, leaving the answer in register 2.

```
SUBROUTINE TIMES
0  MOVI 2,0  ; Move (immediate) a 0 into reg 2
1  JUMPZ 0,5  ; Jump to instruction 5 if reg 0 contains 0
2  ADD 2,1   ; Add contents of reg 1 into that of reg 2
3  SUBI 0,1   ; Subtract (immediate) a 1 from contents of reg 0
4  JUMP 1     ; Jump to instruction 1
5  RET       ; Return from subroutine call
```

Such programs are represented formally by their parse trees. The above program is represented by the constant

```
'(TIMES
  (MOVI 2 0)
  (JUMPZ 0 5)
  (ADD 2 1)
  (SUBI 0 1)
  (JUMP 1)
  (RET)).
```

Programs will be given meaning operationally. In particular we will define a notion of “machine state” and a “state transition” function and then consider the states produced by running programs a given number of instructions.

The state of our machine will be represented by instances of the shell class **ST**.

Shell Definition.

Add the shell **ST** of 5 arguments
with recognizer function **STP**,
accessor functions **PC**, **STK**, **MEM**, **HALTEDP**, and **DEFS**, and
default functions **ZERO**, **ZERO**, **ZERO**, **ZERO**, and **ZERO**.

Intuitively, **DEFS** is a list of programs as above; the “program counter,” **PC**, is a pair, e.g., **(TIMES . 0)**, pointing to some instruction in some program of

DEFS; **STK** is a “subroutine call stack” containing the suspended program counters of the active subroutine calls; **MEM** is a list of the contents of an arbitrary number of consecutive memory locations starting at “address” 0; and **HALTEDP** indicates the status of the computation. If **HALTEDP** is **F** the computation is “proceeding normally;” otherwise we say it has “halted.” We adopt the convention of referring to the first few memory locations as “registers.”

The “state transition” function is

Definition.

```
(STEP S)
=
(IF (HALTEDP S)
  S
  (EXECUTE (FETCH (PC S) (DEFS S)) S)).
```

Thus, **STEP** is a no-op if the state has already halted. Otherwise, we **FETCH** the instruction indicated by the program counter and **EXECUTE** that instruction on the current state. **EXECUTE** is just a big case split, on the “op-code” of the instruction:

Definition.

```
(EXECUTE INS S)
=
(LET ((OP (CAR INS)) ; INS = '(op a b ...)
      (A (CADR INS))
      (B (CADDR INS)))
  (CASE OP
    (MOVE (MOVE A B S))
    (MOVI (MOVI A B S))
    (ADD (ADD A B S))
    ...
    (CALL (CALL A S))
    (RET (RET S))
    (OTHERWISE S))).
```

For each instruction in the programming language we define a function in the logic that takes as its arguments the operands of the instruction and the current state. The function returns the appropriate successor state. The functional version of the **ADD** instruction is given next.

Definition.

```

(ADD ADDR1 ADDR2 S)
=
(ST (ADD1-PC (PC S))
  (STK S)
  (PUT ADDR1
    (PLUS (GET ADDR1 (MEM S))
      (GET ADDR2 (MEM S)))
    (MEM S))
  F
  (DEFS S))

```

The program counter of the resulting state is one greater than that of the given state. The stack is unchanged. The memory is modified by storing at **ADDR1** the sum of the contents of the two operands, **ADDR1** and **ADDR2**. The halt flag is **F**. The program space is unmodified.

In the interests of brevity we will not give the definitions of all other instructions. They can often be inferred from their names and the definition of **ADD**. **MOVE**, for example, is like **ADD** but stores the contents of **ADDR2** into **ADDR1**. **MOVI** (move immediate) is like **MOVE** except stores **ADDR2** (rather than its contents) into **ADDR1**. **CALL** replaces the program counter by a pair that points to the first instruction of the called subroutine, **'(addr1 . 0)**, and pushes onto the stack the result of incrementing the current program counter. **RET** sets the **HALTEDP** flag to **T** if the stack is empty and otherwise restores the program counter from the top of the stack and pops the stack. See example file 16 for details.

Finally, we can define our “small machine” with

Definition.

```

(SM S N)
=
(IF (ZEROP N)
  S
  (SM (STEP S) (SUB1 N))).

```

This function applies **STEP** to the initial state, **S**, **N** times. Since **STEP** is a no-op on halted states, we might have said “This function runs **S** a maximum of **N** steps, or until termination.”

In more realistic languages instructions can cause “errors.” This can be accommodated by permitting the **HALTEDP** flag to take on non-**F** “error codes” as values. Such codes would effectively halt the machine defined above. Then, for example, we might redefine **EXECUTE** to set the halt field to **'ILLEGAL-OPCODE** in its “otherwise clause” above, signaling an error when an unrecognized instruction is encountered. This error would then be signaled if

the program counter were ever incremented “too far” in some program, because the **FETCH** in **STEP** would return the illegal instruction 0. Similarly, we might redefine **ADD** to set the halt field to **'ARITHMETIC-OVERFLOW** if the required sum is “too big,” and **CALL** to signal **'UNDEFINED-LABEL** if the called program were not among the definitions of the state. This approach allows each instruction to signal errors without complicating (or even changing) the machine as defined above.

Many practical languages include instructions that cause unspecified effects on some parts of the state. This happens frequently when one is formalizing an informally described machine or language: the informal text simply may not enumerate the effect of every instruction on every part of the state. For example, suppose the machine in question has a Boolean “condition code,” intended to indicate the arithmetic sign of the result of the most recently completed operation. The informal specification of the arithmetic instructions will probably include a description of how this condition code is set. But the informal specification of, say, the **CALL** instruction, may not. In the absence of more information, a faithful formalization of such a text must somehow leave this field unspecified after **CALL**. At first sight the operational approach here fails, since there is in fact no functional relation between successive states; indeed, one might imagine a whole family of possible successor states any one of which may be chosen. The user of such a machine designs programs that do not depend on which of the allowed successors is actually taken. It is possible to formalize such a language while staying within the operational framework by introducing unspecified functions for “determining” the unspecified parts of the successor state. In the case of the post-**CALL** condition code, we wish to require that the new condition code be Boolean valued but otherwise we do not know anything about it. We might therefore introduce by constraint a Boolean valued function, **POST-CALL-CC**, that takes one argument, a machine state.

Constraint.

Constrain the function **POST-CALL-CC** to have the property

```
(OR (TRUEP (POST-CALL-CC S))
    (FALSEP (POST-CALL-CC S))).
```

A suitable witness is the function that always returns **T**. We could then use **POST-CALL-CC** to specify the value of the condition code after the **CALL** instruction. In this way we actually define a whole family of languages in an operational way. This is an example of the use of constrained functions to avoid “overspecification.” If a formalization “feels too confining” the judicious use of constrained functions may help.

3.6.2. Proving Programs Correct

Having defined the machine we can now turn to correctness theorems for programs. One theorem we can prove is that if **SM** is used to execute a call of the **TIMES** program shown above, on a state containing **I** in register 0 and **J** in register 1, and **SM** is allowed to run $4+4\mathbf{I}$ instructions, then register 2 of the resulting state contains $\mathbf{I}*\mathbf{J}$. This form of “correctness theorem” for **TIMES** appears inadequate in two respects. The first is that it seems to address only the situation in which the call of **TIMES** is given exactly the required amount of time to run. The second is that it does not tell us anything about the resulting state except the contents of register 2. It turns out that the first objection is not serious but the second is. We return to this point in a moment.

We can prove theorems about **SM** that have nothing to do with any particular program. One extremely useful theorem is

Theorem. SM-PLUS
 $(\text{EQUAL } (\text{SM } S (\text{PLUS } I J))$
 $(\text{SM } (\text{SM } S I) J)).$

That is, an execution of length $\mathbf{I}+\mathbf{J}$ can be decomposed into one of length **I** followed by one of length **J**. An equivalent result is that if **N** is a natural number and $\mathbf{N} \geq \mathbf{K}$ then $(\text{SM } S \mathbf{N})$ is the same as running **S** **K** steps and then running $\mathbf{N}-\mathbf{K}$ more steps. This result overcomes the first objection to the correctness theorem for **TIMES**. If **S** is a state poised to execute a call of **TIMES** and $\mathbf{N} \geq 4+4\mathbf{I}$, where **I** is the contents of register 0 in **S**, then $(\text{SM } S \mathbf{N})$ is the same as $(\text{SM } (\text{SM } S 4+4\mathbf{I}) \mathbf{N}-4+4\mathbf{I})$. That is, we can break the run into two parts, the first of which is exactly the “right” length for the **CALL** of **TIMES**.

However, this highlights the second inadequacy in the correctness result for **TIMES**. To use it to prove the correctness of programs that **CALL TIMES**, it is necessary (at least in many cases) to characterize the entire state produced by the **CALL**, not just the “answer.”

We now give a usable correctness theorem for our **TIMES** program. A formal statement of the theorem is given below. We paraphrase and explain it here. First, suppose **S** is a state poised to execute a **CALL** of **TIMES**. By that we mean that the program counter of **S** points to a **'(CALL TIMES)** instruction and that the definition of **TIMES** in **S** is in fact the program constant we exhibited above. Characterizing the state this way permits the theorem to be applied to any call of **TIMES** in any program and permits the list of definitions to be arbitrary as long as it includes our **TIMES** somewhere. Next, suppose that the memory of **S** contains at least three registers. This is necessary since the program uses register 2. Let us call the contents of register 0 and 1 **I** and **J** respectively. Suppose **I** is a natural number. We comment on this condition in a moment. Finally, suppose the halt field of **S** is **F**, i.e., the state is not halted.

Given these conditions, the theorem tells us explicitly what state is produced by running S $4+4I$ steps. In particular, it is the state obtained from S by

- incrementing the program counter by one,
- leaving the stack is unmodified,
- storing a 0 into register 0 and $I*J$ into register 2,
- leaving the halt flag F , and
- leaving the definitions unmodified.

The condition that I be numeric comes from our decision to characterize the new memory as having a 0 in register 0. If I is not a natural number the program terminates without changing register 0.

A formal statement of this correctness result is shown below. We have defined **(TIMES-PROGRAM)** to be the **TIMES** subroutine constant above and **(TIMES-CLOCK I)** to be $4+4I$. We adopt the discipline of abstracting the runtime of each program into a corresponding “clock” function so composition is easier. The beautiful thing about the following result is that it is easy to prove and, in combination with the **SM-PLUS** theorem, permits us in the future to treat **(CALL TIMES)** exactly as we would a primitive instruction, except that the instruction count is decremented by **(TIMES-CLOCK I)** instead of by one.

Theorem. TIMES-CORRECT

```
(IMPLIES
  (AND (EQUAL (FETCH (PC S) (DEFS S))
              '(CALL TIMES))
        (EQUAL (ASSOC 'TIMES (DEFS S))
                (TIMES-PROGRAM))
        (LESSP 2 (LENGTH (MEM S)))
        (EQUAL I (GET 0 (MEM S)))
        (EQUAL J (GET 1 (MEM S)))
        (NUMBERP I)
        (NOT (HALTEDP S)))
    (EQUAL (SM S (TIMES-CLOCK I))
          (ST (ADD1-PC (PC S))
              (STK S)
              (PUT 0 0
                  (PUT 2 (TIMES I J)
                        (MEM S)))
              F
              (DEFS S)))))
```

We call theorems such as that above the “standard correctness” theorem for the subroutine concerned, in this case **TIMES**. The reader interested in how

Nqthm can be led to the proofs of such theorems should see example file 16. That file also illustrates the use of this theorem in the proof of the standard correctness theorem for a program that calls **TIMES**. That is, the file illustrates how standard correctness theorems can be combined.

3.6.3. Termination and Non-Termination

Observe that the standard correctness theorem implicitly establishes that the subroutine terminates: control eventually returns to the instruction after the **CALL** and the stack is unchanged. One might even define “termination” with this condition on the program counter and stack. However, there is an attractively direct statement of termination, that we call the “standard termination” theorem.

The standard termination theorem for some subroutine, **fn**, exhibits an expression, **clock**, which is generally a function of the “input” to **fn**, with the property that a “top-level” execution of the definition of **fn** sets the halt flag in **clock** steps. By a “top-level” execution we mean one starting in a state in which **fn** is defined as desired, the program counter points to the first instruction in the program, **'(fn . 0)**, and this is the “main program,” i.e., the stack is **NIL**. The standard termination theorem is attractive because it establishes that in a certain context, the execution of **fn** actually sets the processor’s halt flag within an explicitly given number of steps. We return to this idea in the next section.

It is also possible to state and (sometimes, at least) prove that some programs do not terminate, or do not terminate under certain conditions on their inputs. In particular, one might prove that, under certain conditions on the state **S**, **(NOT (HALTEDP (SM S N)))**. Since such a theorem is true for all values of **N**, we feel justified in claiming that **S** is a “non-terminating” state.

3.6.4. Proving Theorems about the Language

Because we have formalized the semantics of the programming language it is possible to prove theorems about the language itself, rather than just about particular programs. We have already seen one such theorem,

Theorem. SM-PLUS
(EQUAL (SM S (PLUS I J))
(SM (SM S I) J)).

This theorem puts no restrictions on the program being run.

The following theorem is a non-termination result. It says that if the current program counter in a running state points to a **JUMP** to itself, the state is non-terminating. Again, no particular program is involved.

Theorem.

```
(IMPLIES (AND (NOT (HALTEDP S))
              (EQUAL (FETCH (PC S)) (DEFS S))
              (LIST 'JUMP I))
          (NUMBERP I)
          (EQUAL (CDR (PC S)) I))
          (NOT (HALTEDP (SM S N))))).
```

A final illustration of a general theorem about the **SM** language relates the standard correctness theorem to the standard termination theorem. The theorem may be paraphrased as follows. If the current instruction in a running state is a **CALL** of some program **prog**, and in some non-zero number of steps, **n**, the stack is the same as it was at the **CALL**, then a top-level execution of **prog** sets the halt flag within **n** steps.

Theorem.

```
(IMPLIES (AND (NOT (HALTEDP S))
              (EQUAL (FETCH (PC S)) (DEFS S))
              (LIST 'CALL PROG))
          (NOT (ZEROP N))
          (EQUAL (STK (SM S N)) (STK S)))
          (HALTEDP
           (SM (ST (CONS PROG 0)
                  NIL
                  (MEM S)
                  F
                  (DEFS S))
              N))))).
```

This theorem is difficult to prove because it involves two different executions of the program: the “continuing execution” in state **S** and the “top-level execution” in the state mentioned in the conclusion above. A sketch of the argument is as follows. From the hypotheses we can conclude that the continuing computation executes a return instruction “balancing” the **CALL** within **N** steps. Say this happens at step **k**. (We are not told that **N** is the minimum number of steps necessary to get back to the original stack; indeed, the hypotheses allow for the called program to return immediately and for the computation to proceed, possibly even exiting from the caller before eventually reentering!) Then, with the help of various general theorems about **SM**, which establish that the two computations proceed isomorphically through the program, we can conclude that in the top-level execution, at the **k**–1st step, a return is executed while the

stack is **NIL**. Such a return sets the halt flag and the flag stays set for the remaining $n-(k-1)$ steps of the top-level execution. The proof of this theorem is in example file 16.

3.6.5. Other Approaches

The idea of using constrained functions to avoid overspecification can be generalized. Suppose, for example, we introduce by constraint two functions,

Constraint.

Constrain the functions **STATEP** and **STEP** to have the property

(IMPLIES (STATEP S) (STATEP (STEP S))).

We could then define a “generic interpreter”

Definition.

(INTERP S N)

=

(IF (ZEROP N)

S

(INTERP (STEP S) (SUB1 N)))

and prove some interesting theorems about it, such as

Theorem.

(IMPLIES (STATEP S) (STATEP (INTERP S N)))

Theorem.

**(EQUAL (INTERP S (PLUS I J))
(INTERP (INTERP S I) J)).**

Any particular interpreter whose definition is analogous to that of **INTERP**, i.e., is just the iterated application of some “step” function, enjoys the properties of the generic interpreter. Indeed, the particular properties can be derived from the generic ones by functional instantiation.

Similarly, we might use a combination of constrained and defined functions to introduce states that have a control stack and halt flag but are otherwise unspecified. We might introduce a step function that implements **CALL** and **RET** but leaves unspecified all other state components—with the important exception that none of the other instructions change the control stack or halt flag. We would have thus defined a language with subroutine call and return, without specifying any of the “primitive” operations in the language. We could then prove as generic theorems the theorems relating, say, call and return to top-level

termination or otherwise develop the theory of subroutine call, without having to worry about the particular operations carried out by the language. We could then define some particular language, providing many elaborate primitive operations, and immediately take advantage of the “theory of call and return” without having to rederive it amid the complexities of the actual state and instruction set.

It is possible to relate the operational semantics to other approaches towards formalizing program semantics. In the example file we show how one can prove that particular programs “correspond” to recursively defined functions that carry out the same steps. This is sometimes called the “functional approach” to modeling semantics and was suggested by McCarthy in [98]. For complicated **SM** programs it is often most convenient to first prove that the operational interpretation of the program corresponds to some particular, and usually peculiarly defined, function and then prove that the function has the desired properties. The first step is straightforward because the function in question exactly mimics the execution of the program. But by taking the first step one eliminates **SM** from the problem in the second step.

Also in the example file we illustrate how it is possible to “trade in” the verification conditions generated by a straightforward application of the so-called “inductive assertion” method of Hoare [65] and Floyd [54], for a theorem about the operational interpretation of a given program.

It should be noted that the logic can be used to address program correctness problems without necessarily formalizing the language semantics in the logic. For example, in [26] we present a verification condition generator (*vcg*) for a subset of ANSI Fortran 66 and 77. A verification condition generator is a program that takes two inputs, a program and a specification, and generates formulas, called “verification conditions,” that, if theorems, establish that the program satisfies its specification. The language semantics are embedded in the *vcg* program, and the logic is used simply to express specifications and the verification conditions. We present a *vcg* for a simple language in [24].

3.7. Embedded Formal Systems

An increasingly common use of our system is in the definition of what we will call “interpreters” for other systems. Such systems might be programming languages or computing machines, as discussed above. However, the operational approach is extremely flexible and can be used to deal with a variety of other formal (or formalized) systems. We give six brief examples ranging from an interpreter for a simple physical system, through more obviously formal systems such as hardware design to purely formal systems such as lambda-calculus and set theory.

3.7.1. *Real Time Control*

The simple real time control problem discussed in [20] is to show that a certain control program keeps a vehicle on a straight-line course in a varying crosswind. We formalized the problem with an interpreter that is essentially a simulator for (a drastic simplification of) the physical system. The interpreter takes two arguments. The first is the “state” of the vehicle and the world—the vehicle position and velocity and the wind velocity. The second is a “wind history”—a list of wind velocity increments describing how the wind changes at discrete time units over some finite time interval. The interpreter determines the final state of the vehicle by “simulating” each state change, that is, by calculating for each sampling unit the new wind velocity, the resulting position of the vehicle, the associated sensor reading, the response of the control program, and the effect on the vehicle’s velocity. Two theorems are proved about the simulated vehicle: the vehicle does not wander outside of a certain corridor if the wind changes “smoothly,” and the vehicle homes to the proper course if the wind stays steady for a certain amount of time. For the events necessary to lead the theorem prover to these proofs, see example file 4.

3.7.2. *Games*

Game playing programs raise many of the same issues raised in control problems. Such programs attempt to “control” the board in the face of hostile interaction with a human opponent. Games may be formalized with interpreters that can be thought of as “referees.” Such an interpreter takes a series of boards as its input and announces whether a legal game was played and, if so, what the outcome was. A game playing program is a function which generates the board sequence as a function of the moves of the opponent. Proving that a certain strategy wins (or never loses) is very similar to proving some control loops: an invariant is identified and shown to be maintained by the strategy and to result in a win in the terminal state. This proof method is applied to the game of Nim, where the two players alternately remove sticks from various piles, in example file 60. In file 17 we prove that a certain strategy for playing tic-tac-toe never loses. That proof is interesting because it uses a brute force method, forcing Nqthm to consider all possible games. Specifying game playing programs is surprisingly subtle. Intuitively, one merely wishes to say that the game playing program generates a board sequence that is approved by the referee and ends in a win, say, for the program. But this ignores many issues. For example, the game playing program must respect the opponent’s moves: the resulting sequence of boards must accurately reflect the moves by the opponent.

Imagine a game playing program that refused to put your piece on the specified square because it upset the program's plan! But this respect for the opponent's moves means that not all games generated are legal. A more accurate specification must say that if an illegal game is generated then it is the fault of the opponent. Another problem is that the game playing program cannot "look ahead" into the opponent's moves, i.e., it cannot prompt the opponent for two moves before replying to the first. We deal with these problems for tic-tac-toe in example file 17. These kind of problems also arise when one tries to formalize the interaction between a control program and the environment. Such problems are easier to study in the setting of games because the rules of engagement are well understood.

3.7.3. *Towers of Hanoi*

The Towers of Hanoi is a child's puzzle with which you are probably familiar. It is played on a board with three pegs and a stack of disks each of which has a peg-sized hole in the center and a diameter different from that of all other disks. Initially the disks are arranged in a stack on a single peg so that no disk sits atop one that is smaller. The object of the game is to move the disks one at a time so as to reassemble the initial stack on one of the two initially empty pegs without ever placing a disk atop one that is smaller. Writing a computer program to "solve" this problem is a typical exercise in programming language courses when introducing recursion. We wished to write a function that solves the problem, and we wished to prove that it is correct.

In our formalization, the function called **HANOI** takes as input three pegs, **A**, **B**, and **C**, and a number **n** and generates a list of "moves" that supposedly is sufficient to move a tower of **n** disks from peg **A** to peg **C** using peg **B** for temporary storage. A "move" is just a pair $\langle i, j \rangle$ whose interpretation is that the top disk on peg **i** is to be moved to peg **j**. To state the correctness of **HANOI** we defined an "interpreter" for a list of moves. The interpreter takes as arguments a "state" and a "move list" and returns either the final state or **F** indicating that an illegal move was attempted. A state is simply a triple of lists, each list representing the configuration of a given peg by showing the diameters of the disks on that peg. The interpreter is defined to carry out each move on the state successively, checking each time that the designated source peg is non-empty and that its top disk is smaller than the top disk, if any, on the designated destination peg. The theorem proved was that if the initial state was as expected—i.e., there is a well-formed tower on peg **A** and the other pegs are empty—then the final state, after playing out the moves delivered by **(HANOI A B C N)**, is non-**F** and configured as desired—i.e., the tower is on peg **C** and

the other pegs are empty.⁸ The assurance that **HANOI** never put a disk on top of a smaller one is not stated explicitly in the theorem but rather follows from the definition of the interpreter.

3.7.4. *Hardware Verification*

Combinational logic can be modeled by propositional expressions. That is, (**AND** **S1** **S2**) can be thought of as a circuit that takes two input bits, **S1** and **S2**, and delivers their logical **AND**. However, consider modeling a device, such as a cpu, that contains state-holding components that are changed over time by combinational logic. Such a model of a microcoded cpu is presented in [73, 70]. Essentially a register-transfer model is used, except the model is formalized as a recursively defined function. All but one of the arguments of the function represent state-holding components in the device, such as the registers of the processor, the microaddress register, the memory, etc. One argument is an oracle that represents the values found on various input lines at each clock cycle over some time interval, as well as “nondeterministic” parameters determining memory response time. The definition of the function is such that if the oracle is exhausted, the entire state is returned. Otherwise, the function recurses, using combinational logic expressions (and, where appropriate, input from the oracle) to determine the new value of every state holding component. The theorem proved about this machine is that, under a certain abstraction, it is equivalent to a machine-code interpreter. In [74], the formalization of a fragment of a commercial hardware description language is described. The formalization is an interpreter which takes as one of its arguments a “netlist” describing the interconnection of a collection of “modules.” Some of the modules are primitive and others are defined. Another argument to the interpreter specifies initial values on input lines. The interpreter delivers both the values on the output lines and the values of the “state holding devices” in the netlist. Functions are also provided for determining the fanout, gate-delay, and other aspects of the design. An extensive library of lemmas was developed to allow hierarchically composed proofs about designs. The hardware description language has been used to describe the design of a microprocessor, the FM9001, and to prove that the design implemented a machine language interpreter. The FM9001 was later fabricated from the design.

⁸Actually, the statement of the theorem is rather delicate because one must know that **A**, **B**, and **C** are distinct peg numbers. The proof was carried out by Matt Kaufmann.

3.7.5. *Lambda Calculus*

The Church-Rosser theorem for lambda calculus says that if a lambda-expression x can be reduced to two expressions y and z then there is a w such that both y and z reduce to w . Briefly, the formalization of the lambda calculus used in [128, 127] proceeds along the following lines. Represent lambda-expressions as list or shell constants, e.g., `'(LAMBDA (X) (X X))` is such a constant. Define what is meant by α - and β -steps by defining predicates that determine whether one lambda-expression is obtained from another by the appropriate transformation. Define the concept that one lambda-expression *reduces* to another via a given finite sequence of lambda-expressions, each obtained from the previous by an α - or β -step. The theorem is then: if \mathbf{X} , \mathbf{Y} , and \mathbf{Z} are lambda-expressions and \mathbf{X} reduces to \mathbf{Y} via the sequence $\mathbf{S1}$ and \mathbf{X} reduces to \mathbf{Z} via the sequence $\mathbf{S2}$ then there exist a lambda-expression \mathbf{W} and sequences $\mathbf{S3}$ and $\mathbf{S4}$ such that \mathbf{Y} reduces to \mathbf{W} via $\mathbf{S3}$, and \mathbf{Z} reduces to \mathbf{W} via $\mathbf{S4}$. The existential quantifiers are then eliminated by defining functions for constructing the alleged \mathbf{W} , $\mathbf{S3}$, and $\mathbf{S4}$. For the complete details see example file 63.

Once again we see a trade-off between difficulty of statement and difficulty of proof: key steps in the proof, namely, the construction of \mathbf{W} , $\mathbf{S3}$, and $\mathbf{S4}$, are taken during the formalization of the problem.

3.7.6. *An Incompleteness Theorem*

To state Gödel's incompleteness theorem it was necessary to formalize within our logic another logic. The object logic was Shoenfield's first order logic with Cohen's axioms for hereditarily finite set theory, $Z2$. The formalization was done by representing *formulas* in the object logic by list- and shell-constants and then defining within our logic a proof checker for the object logic. That is, the function defined the concept that \mathbf{P} is a *proof* of \mathbf{X} . The theorem was then formalized by saying that there exists a closed formula, \mathbf{X} , of the object logic such that if $\mathbf{P1}$ is a proof of \mathbf{X} or $\mathbf{P2}$ is a proof of its negation, then there exist proofs $\mathbf{P3}$ and $\mathbf{P4}$ of both \mathbf{X} and its negation. The existential quantifiers are again removed by defining the appropriate functions. The details are given in [127, 129] and in example file 64.

4 A Precise Description of the Logic

4.1. Apologia

Traditionally, logicians keep their formal systems as simple as possible. This is desirable because logicians rarely use the formal systems themselves. Instead, they stay in the metatheory and (informally) prove results about their systems.

The system described here is intended to be used to prove interesting theorems. Furthermore, we want to offer substantial mechanical aid in constructing proofs, as a means of eliminating errors. These pragmatic considerations have greatly influenced the design of the logic itself.

The set of variable and function symbols is influenced by conventions of half a dozen different Lisp systems. The set of axioms is unnecessarily large from the purely logical viewpoint. For example, it includes axioms for both the natural numbers and ordered pairs as distinct classes of objects. We found certain formalization problems cumbersome when one of these classes is embedded in the other, as is common in less pragmatically motivated logics. Furthermore, the distinction between the two classes makes it easier for us to provide mechanical assistance appropriate to the particular domain. Similarly, a general purpose quantifier over finite domains is provided, even though recursively defined functions suffice. We provide an interpreter for the logic in the logic—at the cost of complicating both the notation for constants and the axioms to set up the necessary correspondence between objects in the logic and the term structure of the language—so that certain useful forms of metatheoretic

reasoning can be carried out in the logic. Our induction principle is very complicated in comparison to those found in many other logics; but it is designed to be directly applicable to many problems and to produce simple proofs.

To achieve our goal of providing assistance in the proof of interesting theorems it must be possible, indeed, convenient, to *state* interesting theorems. This requires that we allow the user to extend the logic to introduce new objects and concepts. Logically speaking, the main theorem and all the intermediate lemmas are proved in the single final theory. But practically speaking, the theory “evolves” over time as the user repeatedly extends it and derives intermediate results. We provide three “extension” principles—the “shell principle” for introducing new, inductively-defined data types, the “definitional principle” for defining new recursive functions, and the “constraint principle” for introducing functions that are undefined but constrained to have certain properties. Our extension principles, while complicated, are designed to be sound, easy to use and to mechanize, and helpful in guiding the discovery of proofs.

While the logic is complicated compared to most mathematical logics, it is simple compared to most programming languages and many specification languages. If our presentation of it makes it seem “too complicated” it is perhaps merely that we are presenting all of the details.

4.2. Outline of the Presentation

In presenting our logic we follow the well-established tradition of incremental extension. We begin by defining a very simple syntax, called the *formal syntax*, of the language. A much richer syntax, called the *extended syntax*, which contains succinct abbreviations for constants such as numbers, lists, and trees, is defined only after we have axiomatized the primitive data types of the logic.

Using the formal syntax we present the axioms and rules of inference for propositional calculus with equality, the foundation of our theory. Next we embed propositional calculus and equality in the term structure of the logic by defining functional analogues of the propositional operators. We then present the shell principle and use it to add the axioms for natural numbers, ordered pairs, literal atoms, and negative integers.

At this point we have enough formal machinery to explain and illustrate the extended formal syntax.

We then present our formalization of the ordinals up to ϵ_0 . The “less-than” relation on these ordinals plays a crucial role in our principles of mathematical induction and recursive definition.

Next we add axioms defining many useful functions. Then we embed the semantics of the theory in the theory by axiomatizing an interpreter for the logic

as a function. In order to do this it is necessary to set up a correspondence between the terms in the formal syntax and certain constants in the logic, called the “quotations” of those terms. Roughly speaking, the quotation of a term is a constant in the logic whose value under the interpreter is equal to the term.

We complete the set of axioms by defining our general purpose quantifier function, which, much like the “mapping” functions of Lisp, includes among its arguments objects denoting terms which are evaluated with the interpreter.

Finally, we state the principles of inductive proof and recursive definition.

We frequently pause during the presentation to illustrate the concepts discussed. However, we do not attempt to motivate the development or explain “how” certain functions “work” or the role they play in the subsequent development. We assume that the reader interested in such asides will have first read the primer, Chapter 2. Familiarity with the primer is completely unnecessary to follow the precise development of the theory.

We classify our remarks into eight categories:

- **Terminology:** In paragraphs with this label we define syntactic notions that let us state our axioms or syntactic conventions precisely. The concept defined is *italicized*.
- **Abbreviation:** In paragraphs with this label we extend the previously agreed upon syntax by explaining how some string of characters is, henceforth, to be taken as shorthand for another.
- **Example:** We illustrate most of the terminology and abbreviations in paragraphs with this label. Technically, these paragraphs contain no new information, but they serve as a way for the reader to check his or her understanding.
- **Axiom or Defining Axiom:** A formula so labelled is an axiom of our system. Axioms of the latter sort are distinguished because they uniquely define a function.
- **Shell Definition:** A paragraph so labelled schematically specifies a set of axioms of our system.
- **Extension Principle:** A paragraph so labelled describes a principle which permits the sound introduction of new function symbols and axioms.
- **Rule of Inference:** A paragraph so labelled describes a rule of inference of our system.
- **Note:** Assorted remarks, such as alternative views, are collected in paragraphs with this label.

4.3. Formal Syntax

Terminology. A finite sequence of characters, s , is a *symbol* if and only if s is nonempty, each character in s is a member of the set

```
{A B C D E F G H I J K L M
 N O P Q R S T U V W X Y Z
 0 1 2 3 4 5 6 7 8 9
 $ ^ & * _ - + = ~ { } ? < >},
```

and the first character of s is a letter, i.e., in the set

```
{A B C D E F G H I J K L M
 N O P Q R S T U V W X Y Z}.
```

Examples. **PLUS**, **ADD1**, and **PRIME-FACTORS** are symbols. ***1*TRUE**, **123**, **A/B**, and **#.FOO** are not symbols.

Terminology. The *variable symbols* and *function symbols* of our language are the symbols other than **T**, **F**, and **NIL**.

Terminology. Associated with every function symbol is a nonnegative integer called the *arity* of the symbol. The arity indicates how many argument terms must follow each application of the function symbol. The arity of each function symbol in the Ground Zero logic is given in the table below. We also include brief descriptive comments in the hopes that they will make subsequent examples more meaningful.

Table 4.1

symbol	arity	comment
ADD1	1	successor function for natural numbers
ADD-TO-SET	2	adds an element to a list if not present
AND	2	logical and
APPEND	2	list concatenation
APPLY-SUBR	2	application of primitive fn to arguments
APPLY\$	2	application of fn to arguments
ASSOC	2	association list lookup
BODY	1	body of a fn definition
CAR	1	first component of ordered pair
CDR	1	second component of ordered pair
CONS	2	constructs ordered pairs
COUNT	1	size of an object

DIFFERENCE	2	natural difference of two natural numbers
EQUAL	2	equality predicate
EVAL\$	3	interpreter for the logic
FALSE	0	false object
FALSEP	1	predicate for recognizing FALSE
FIX	1	coerces argument to 0 if not numeric
FIX-COST	2	increments cost if argument is non- F
FOR	6	general purpose quantifier
FORMALS	1	list of formal arguments of a function
GEQ	2	greater than or equal on natural numbers
GREATERP	2	greater than on natural numbers
IDENTITY	1	identity function
IF	3	if-then-else
IFF	2	if and only if
IMPLIES	2	logical implication
LEQ	2	less than or equal on natural numbers
LESSP	2	less than on natural numbers
LISTP	1	recognizes ordered pairs
LITATOM	1	recognizes literal atoms
MAX	2	maximum of two natural numbers
MEMBER	2	membership predicate
MINUS	1	constructs negative of a natural number
NEGATIVEP	1	recognizes negatives
NEGATIVE-GUTS	1	absolute value of a negative
NLISTP	1	negation of LISTP
NOT	1	logical negation
NUMBERP	1	recognizes natural numbers
OR	2	logical or
ORDINALP	1	recognizes ordinals
ORD-LESSP	2	less than on ordinals up to ϵ_0
PACK	1	constructs a LITATOM from ASCII codes
PAIRLIST	2	pairs corresponding elements
PLUS	2	sum of two natural numbers
QUANTIFIER-INITIAL-VALUE	1	initial value of a quantifier
QUANTIFIER-OPERATION	3	operation performed by quantifier
QUOTIENT	2	natural quotient of two natural numbers
REMAINDER	2	mod
STRIP-CARS	1	list of CARS of argument list
SUB1	1	predecessor function on natural numbers
SUBRP	1	recognizes primitive function symbols
SUM-CDRS	1	sum of CDRS of elements of argument list
TIMES	2	product of two natural numbers

TRUE	0	true object
TRUEP	1	recognizes TRUE
UNION	2	union of two lists
UNPACK	1	explodes a LITATOM into ASCII codes
V&C\$	3	determines value and cost of an expr
V&C-APPLY\$	2	determines value and cost of fn application
ZERO	0	0
ZEROP	1	recognizes 0 and nonnatural numbers

The arity of each user-introduced function symbol is declared when the symbol is first used as a function symbol.

Terminology. A *term* is either a variable symbol or else is a sequence consisting of a function symbol of arity n followed by n terms.

Note. Observe that we have defined a term as a tree structure rather than a character sequence. Of interest is how we display such trees.

Terminology. To *display* a symbol, we merely write down the characters in it. To *display* a term that is a variable symbol, we display the symbol. To *display* a nonvariable term with function symbol **fn** and argument terms t_1, \dots, t_n , we write down an open parenthesis, a display of **fn**, a nonempty string of spaces and/or carriage returns, a display of t_1 , a nonempty string of spaces and/or carriage returns, ..., a display of t_n and a close parenthesis.

Note. In the first edition of this handbook, we permitted comments in the display of formal terms. In the second edition, we have disallowed comments in formal terms. Comments are permitted in the extended syntax, which is the syntax used throughout the book (once that syntax is defined) and in the implementation. Thus, the only effect of this change is to delay the precise description of the comment convention until we present the extended syntax.

Examples. The following are (displays of) terms:

```
(ZERO)

(ADD1 X)

(PLUS (ADD1 X) (ZERO))

(IF B (ZERO) (ADD1 X))

(IF B (ZERO)
  (ADD1 X))
```

Terminology. Our axioms are presented as formulas in propositional calculus with equality. The formulas are constructed from terms, as defined above, using the equality symbol and the symbols for “or” and “not”. More precisely, an *atomic formula* is any string of the form $t_1=t_2$, where t_1 and t_2 are terms. A *formula* is either an atomic formula, or else of the form $\neg(\text{form1})$, where **form1** is a formula, or else of the form $(\text{form1} \vee \text{form2})$, where **form1** and **form2** are both formulas. Parentheses are omitted when no ambiguity arises.

Abbreviations. We abbreviate $\neg(t_1 = t_2)$ by $(t_1 \neq t_2)$. If **form1** and **form2** are formulas then $(\text{form1} \rightarrow \text{form2})$ is an abbreviation for $(\neg(\text{form1}) \vee \text{form2})$ and $(\text{form1} \wedge \text{form2})$ is an abbreviation for $\neg(\neg(\text{form1}) \vee \neg(\text{form2}))$.

4.3.1. Terminology about Terms

Terminology. To talk about terms, it is convenient to use so-called “metavariables” that are understood by the reader to stand for certain variables, function symbols, or terms. In this book we use lower case words to denote metavariables.

Example. If **f** denotes the function symbol **PLUS**, and **t** denotes the term **(ADD1 Y)**, then $(f\ t\ X)$ denotes the term **(PLUS (ADD1 Y) X)**.

Terminology. If **i** is a nonnegative integer, then we let **Xi** denote the variable symbol whose first character is **X** and whose other characters are the decimal representation of **i**.

Example. If **i** is **4**, **Xi** is the variable symbol **X4**.

Terminology. A term **t** is a *call* of **fn** with *arguments* a_1, \dots, a_n iff **t** has the form $(fn\ a_1 \dots a_n)$.

Terminology. If a term **t** is a call of **fn** we say **fn** is the *top function symbol* of **t**. A function symbol **fn** is *called in* a term **t** iff either **t** is a call of **fn** or **t** is a nonvariable term and **fn** is called in an argument of **t**. The set of *subterms* of a term **t** is $\{t\}$ if **t** is a variable symbol and otherwise is the union of $\{t\}$ together with the union of the subterms of the arguments of **t**. The *variables* of a term **t** is the set of variable subterms of **t**.

Examples. The term $(PLUS\ X\ Y)$ is a call of **PLUS** with arguments **X** and **Y**. **PLUS** is called in $(IF\ A\ (PLUS\ X\ Y)\ B)$. The set of subterms of $(PLUS\ X\ Y)$ is $\{(PLUS\ X\ Y)\ X\ Y\}$. The set of variables of $(PLUS\ X\ Y)$ is $\{X\ Y\}$.

4.3.2. Terminology about Theories

Notes. Theories evolve over time by the repeated application of extension principles. For example, to construct our logic we start with propositional calculus with equality and extend it by adding the axioms for the natural numbers. Then we extend it again to get ordered pairs and again to get symbols... We eventually start adding axioms defining functions such as Peano sum, product, etc. When we stop, the user of the theorem prover starts by invoking the extension principles to add his or her own data types and concepts.

Each extension principle preserves the consistency of the original logic, provided certain “admissibility” requirements are met. In order to describe these requirements it is necessary that we be able to talk clearly about the sequence of steps used to create the “current” extension.

Terminology. Formula t *can be proved directly from* a set of axioms A if and only if t may be derived from the axioms in A by applying the rules of inference of propositional calculus with equality and instantiation (see page 127) and the principle of induction (see page 179).

Terminology. There are four kinds of *axiomatic acts*: (a) an application of the shell principle (page 131), (b) an application of the principle of definition (page 180), (c) an application of the constraint principle, and (d) the addition of an arbitrary formula as an axiom.

Terminology. Each such act *adds* a set of axioms. The axioms added by an application of the first three acts are described in the relevant subsections. The axioms added by the addition of an arbitrary formula is the singleton set consisting of the formula.

Terminology. A *history* h is a finite sequence of axiomatic acts such that either (a) h is empty or (b) h is obtained by concatenating to the end of a history h' an axiomatic act that is “admissible” under h' . An arbitrary axiom is admissible under any h' . The specification of the shell and definitional principles define “admissibility” in those instances.

Terminology. The *axioms* of a history h is the union of the axioms added by each act in h together with the axioms described in this chapter.

Terminology. We say a formula \mathbf{t} is a *theorem* of history h iff \mathbf{t} can be proved directly from the axioms of h .

Terminology. A function symbol \mathbf{fn} is *new* in a history h iff \mathbf{fn} is called in no axiom of h (except for the propositional, reflexivity, and equality axioms of page 128), \mathbf{fn} is not a **CAR/CDR** symbol (see below), and \mathbf{fn} is not in the set **{CASE COND F LET LIST LIST* NIL QUOTE T}**.

Terminology. We say a symbol \mathbf{fn} is a *CAR/CDR symbol* if there are at least three characters in \mathbf{fn} , the first character in \mathbf{fn} is **C**, the last character is **R**, and each other character is either an **A** or a **D**.

Examples. The symbol **CADDR** is a **CAR/CDR** symbol. We will eventually introduce an abbreviation that “defines” such symbols to stand for nests of **CARS** and **CDRs**. Because **CADDR** is a **CAR/CDR** symbol it is not new. The definitional principle requires that the function symbol defined be “new.” Hence, it is impossible to define **CADDR**. Similarly, it is impossible to define nine other perfectly acceptable symbols, **CASE**, **COND**, **F**, **LET**, **LIST**, **LIST***, **NIL**, **QUOTE**, and **T**. All of these prohibited symbols will be involved in our abbreviation conventions.

4.4. Embedded Propositional Calculus and Equality

Notes. Our logic is a quantifier-free first order extension of propositional calculus with equality, obtained by adding axioms and rules of inference. Any classical formalization of propositional calculus and equality will suit our purposes. So that this book is self-contained we have included as the first subsection below, one such formalization, namely that of Shoenfield [130].

We then add axioms to define the functional analogues of the propositional operators and the equality relation. This effectively embeds propositional calculus and equality into the term structure of the logic. That is, we can write down and reason about terms that contain propositional connectives, equality, and case analysis. For example, we can write

```
(IF (EQUAL N 0)
  1
  (TIMES N (FACT (SUB1 N))))
```

which is a term equal to 1 if **N** is 0 and equal to (**TIMES N (FACT (SUB1 N))**) otherwise. The ability to write such terms is very convenient later when we begin defining recursive functions.

4.4.1. Propositional Calculus with Equality

Shoenfield's system consists of one axiom schema and four inference rules. A *Propositional Axiom* is any formula of the form

Axiom Schema.

$(\neg(\mathbf{a}) \vee \mathbf{a}).$

The four rules of inference are

Rules of Inference.

Expansion: derive $(\mathbf{a} \vee \mathbf{b})$ from **b**;

Contraction: derive **a** from $(\mathbf{a} \vee \mathbf{a})$;

Associativity: derive $((\mathbf{a} \vee \mathbf{b}) \vee \mathbf{c})$ from $(\mathbf{a} \vee (\mathbf{b} \vee \mathbf{c}))$; and

Cut: derive $(\mathbf{b} \vee \mathbf{c})$ from $(\mathbf{a} \vee \mathbf{b})$ and $(\neg(\mathbf{a}) \vee \mathbf{c})$.

To formalize equality we also use Shoenfield's approach, which involves three axiom schemas. A *Reflexivity Axiom* is any formula of the form

Axiom Schema.

$(\mathbf{a} = \mathbf{a}).$

For every function symbol **fn** of arity **n** we add an *Equality Axiom for fn*.

Axiom Schema.

$$\begin{aligned} &((\mathbf{X1}=\mathbf{Y1}) \rightarrow \\ &(\dots \rightarrow \\ &((\mathbf{Xn}=\mathbf{Yn}) \rightarrow \\ &(\mathbf{fn} \mathbf{X1} \dots \mathbf{Xn}) = (\mathbf{fn} \mathbf{Y1} \dots \mathbf{Yn}))) \end{aligned}$$

Finally, we add

Axiom.

$((\mathbf{X1}=\mathbf{Y1}) \rightarrow ((\mathbf{X2}=\mathbf{Y2}) \rightarrow ((\mathbf{X1}=\mathbf{X2}) \rightarrow (\mathbf{Y1}=\mathbf{Y2}))))).$

This axiom is the only instance we need of Shoenfield's "equality axiom (schema) for predicates."

Note. Finally, we add the rule of inference that any instance of a theorem is a theorem. To make this precise we first define substitution.

Terminology. A finite set s of ordered pairs is said to be a *substitution* provided that for each ordered pair $\langle \mathbf{v}, \mathbf{t} \rangle$ in s , \mathbf{v} is a variable, \mathbf{t} is a term, and no other member of s has \mathbf{v} as its first component. The *result of substituting* a substitution s *into* a term or formula \mathbf{x} (denoted \mathbf{x}/s) is the term or formula obtained by simultaneously replacing, for each $\langle \mathbf{v}, \mathbf{t} \rangle$ in s , each occurrence of \mathbf{v} as a variable in \mathbf{x} with \mathbf{t} . We sometimes say \mathbf{x}/s is the *result of instantiating* \mathbf{x} with s . We say that \mathbf{x}' is an *instance* of \mathbf{x} if there is a substitution s such that \mathbf{x}' is \mathbf{x}/s .

Example. If s is $\{\langle \mathbf{X}, (\text{ADD1 } \mathbf{Y}) \rangle \langle \mathbf{Y}, \mathbf{Z} \rangle \langle \mathbf{G}, \text{FOO} \rangle\}$ then s is a substitution. If \mathbf{p} is the term

$$(\text{PLUS } \mathbf{X} (\text{G } \mathbf{Y} \mathbf{X}))$$

then \mathbf{p}/s is the term

$$(\text{PLUS } (\text{ADD1 } \mathbf{Y}) (\text{G } \mathbf{Z} (\text{ADD1 } \mathbf{Y})))$$

Note that even though the substitution contains the pair $\langle \mathbf{G}, \text{FOO} \rangle$ the occurrence of \mathbf{G} in \mathbf{p} was not replaced by FOO since \mathbf{G} does not occur as a variable in \mathbf{p} .

Rule of Inference. *Instantiation:*
Derive \mathbf{a}/s from \mathbf{a} .

4.4.2. The Axioms for TRUE, FALSE, IF, and EQUAL

Abbreviation. We will abbreviate the term (TRUE) with the symbol \mathbf{T} and the term (FALSE) with the symbol \mathbf{F} .

Axiom 1.

$$\mathbf{T} \neq \mathbf{F}$$

Axiom 2.

$$\mathbf{X} = \mathbf{Y} \rightarrow (\text{EQUAL } \mathbf{X} \mathbf{Y}) = \mathbf{T}$$

Axiom 3.

$$\mathbf{X} \neq \mathbf{Y} \rightarrow (\text{EQUAL } \mathbf{X} \mathbf{Y}) = \mathbf{F}$$

Axiom 4.

$$\mathbf{X} = \mathbf{F} \rightarrow (\text{IF } \mathbf{X} \mathbf{Y} \mathbf{Z}) = \mathbf{Z}$$

Axiom 5.

$X \neq F \rightarrow (IF\ X\ Y\ Z) = Y.$

4.4.3. The Propositional Functions

Defining Axiom 6.

$(TRUEP\ X) = (EQUAL\ X\ T)$

Defining Axiom 7.

$(FALSEP\ X) = (EQUAL\ X\ F)$

Defining Axiom 8.

$(NOT\ P)$
 $=$
 $(IF\ P\ F\ T)$

Defining Axiom 9.

$(AND\ P\ Q)$
 $=$
 $(IF\ P\ (IF\ Q\ T\ F)\ F)$

Defining Axiom 10.

$(OR\ P\ Q)$
 $=$
 $(IF\ P\ T\ (IF\ Q\ T\ F))$

Defining Axiom 11.

$(IMPLIES\ P\ Q)$
 $=$
 $(IF\ P\ (IF\ Q\ T\ F)\ T)$

Abbreviation. When we refer to a term t as a formula, one should read in place of t the formula $t \neq F$.

Example. The term

$(IMPLIES\ (AND\ (P\ X)\ (Q\ Y))\ (R\ X\ Y)),$

if used where a formula is expected (e.g., in the allegation that it is a theorem), is to be read as

$(IMPLIES\ (AND\ (P\ X)\ (Q\ Y))\ (R\ X\ Y)) \neq F.$

Given the foregoing axioms and the rules of inference of propositional calculus

and equality, the above formula can be shown equivalent to

$$((P\ X) \neq F \wedge (Q\ Y) \neq F) \rightarrow (R\ X\ Y) \neq F$$

which we could abbreviate

$$((P\ X) \wedge (Q\ Y)) \rightarrow (R\ X\ Y).$$

Note. The definitional principle, to be discussed later, permits the user of the logic to add new defining axioms under admissibility requirements that ensure the unique satisfiability of the defining equation. The reader may wonder why we did not invoke the definitional principle to add the defining axioms above—explicitly eliminating the risk that they render the system inconsistent. In fact, we completely avoid use of the definitional principle in this presentation of the logic. There are two reasons. First, the definitional principle also adds an axiom (the non-**SUBRP** axiom) that connects the defined symbol to the interpreter for the logic—an axiom we do not wish to have for the primitives. Second, the admissibility requirements of the definitional principle are not always met in the development of the logic.

4.5. The Shell Principle and the Primitive Data Types

Note. The shell principle permits the extension of the logic by the addition of a set of axioms that define a new data type. Under the conditions of admissibility described, the axioms added are guaranteed to preserve the consistency of the logic. The axioms are obtained by instantiating a set of axiom schemas described here. In order to describe the axiom schemas it is first necessary to establish several elaborate notational conventions. We then define the shell principle precisely. Then we invoke the shell principle to obtain the axioms for the natural numbers, the ordered pairs, the literal atoms, and the negative integers.

4.5.1. Conventions

Terminology. We say \mathbf{t} is the **fn nest around \mathbf{b} for \mathbf{s}** iff \mathbf{t} and \mathbf{b} are terms, **fn** is a function symbol of arity 2, \mathbf{s} is a finite sequence of terms, and either (a) \mathbf{s} is empty and \mathbf{t} is \mathbf{b} or (b) \mathbf{s} is not empty and \mathbf{t} is $(\mathbf{fn}\ \mathbf{t}_1\ \mathbf{t}_2)$ where \mathbf{t}_1 is the first element of \mathbf{s} and \mathbf{t}_2 is the **fn nest around \mathbf{b} for the remaining elements of \mathbf{s}** . When we write $(\mathbf{fn}\ \mathbf{t}_1\ \dots\ \mathbf{t}_n) \otimes \mathbf{b}$ where a term is expected, it is an abbreviation for the **fn nest around \mathbf{b} for $\mathbf{t}_1, \dots, \mathbf{t}_n$** .

Note. In the first edition we used “ $(\mathbf{fn} \ t_1 \ \dots \ t_n)@b$ ” to denote what we now denote with “ $(\mathbf{fn} \ t_1 \ \dots \ t_n)\otimes b$.” We changed from “@” to “ \otimes ” in the second edition because the “@” character is now permitted to occur in the extended syntax, in conjunction with backquote notation.

Examples. The **OR** nest around **F** for **A**, **B**, and **C** is the term $(\mathbf{OR} \ A \ (\mathbf{OR} \ B \ (\mathbf{OR} \ C \ F)))$, which may also be written $(\mathbf{OR} \ A \ B \ C)\otimes F$.

Terminology. Each application of the shell principle introduces several “new” function symbols. The invocation explicitly names one symbol as the *constructor* and another as the *recognizer*. Zero or more other symbols are named as *accessors*, and one may be named as the *base* function symbol for that shell.

Terminology. The *constructor function symbols* of a history h consists exactly of the constructor function symbols of applications of the shell principle in h . The *recognizer function symbols* of a history h is the union of $\{\mathbf{TRUEP} \ \mathbf{FALSEP}\}$ with the set consisting exactly of the recognizer function symbols of the applications of the shell principle in h . The *base function symbols* of a history h is the union of $\{\mathbf{TRUE} \ \mathbf{FALSE}\}$ with the set consisting exactly of the base function symbols of the applications of the shell principle in h for which a base function symbol was supplied.

Terminology. We say \mathbf{r} is the *type* of \mathbf{fn} iff either (a) \mathbf{r} is given as the type of \mathbf{fn} in Table 4.2 or (b) \mathbf{fn} is a constructor or base function symbol introduced in the same axiomatic act in which \mathbf{r} was the recognizer function symbol.

Table 4.2

\mathbf{fn}	type of \mathbf{fn}
\mathbf{TRUE}	\mathbf{TRUEP}
\mathbf{FALSE}	\mathbf{FALSEP}

Terminology. A *type restriction* over a set of function symbols s is a nonempty finite sequence of symbols where the first symbol is either the word **ONE-OF** or **NONE-OF** and each of the remaining is an element of s .

Terminology. A function symbol \mathbf{fn} *satisfies* a type restriction $(\mathbf{flg} \ s_1 \ \dots \ s_n)$ iff either \mathbf{flg} is **ONE-OF** and \mathbf{fn} is among the s_i or \mathbf{flg} is **NONE-OF** and \mathbf{fn} is not among the s_i .

Terminology. We say \mathbf{t} is the *type restriction term* for a type restriction (\mathbf{flg} $\mathbf{r}_1 \dots \mathbf{r}_n$) and a variable symbol \mathbf{v} iff \mathbf{flg} is **ONE-OF** and \mathbf{t} is $(\mathbf{OR} (\mathbf{r}_1 \mathbf{v}) \dots (\mathbf{r}_n \mathbf{v})) \otimes \mathbf{F}$ or \mathbf{flg} is **NONE-OF** and \mathbf{t} is $(\mathbf{NOT} (\mathbf{OR} (\mathbf{r}_1 \mathbf{v}) \dots (\mathbf{r}_n \mathbf{v})) \otimes \mathbf{F})$.

Examples. Let \mathbf{tr}_1 be $(\mathbf{ONE-OF LISTP LITATOM})$. Then \mathbf{tr}_1 is a type restriction over the set $\{\mathbf{NUMBERP LISTP LITATOM}\}$. The function symbol **LISTP** satisfies \mathbf{tr}_1 but the function symbol **NUMBERP** does not. The type restriction term for \mathbf{tr}_1 and $\mathbf{x1}$ is $(\mathbf{OR} (\mathbf{LISTP X1}) (\mathbf{OR} (\mathbf{LITATOM X1}) \mathbf{F}))$. Let \mathbf{tr}_2 be $(\mathbf{NONE-OF NUMBERP})$. Then \mathbf{tr}_2 is a type restriction over the set $\{\mathbf{NUMBERP LISTP LITATOM}\}$. The function symbol **LISTP** satisfies \mathbf{tr}_2 but the function symbol **NUMBERP** does not. The type restriction term for \mathbf{tr}_2 and $\mathbf{x2}$ is $(\mathbf{NOT} (\mathbf{OR} (\mathbf{NUMBERP X2}) \mathbf{F}))$.

4.5.2. The Shell Principle

Extension Principle. Shell Principle

The axiomatic act

Shell Definition .

Add the shell **const** of \mathbf{n} arguments
with (optionally, base function **base**,)
recognizer function **r**,
accessor functions $\mathbf{ac}_1, \dots, \mathbf{ac}_n$,
type restrictions $\mathbf{tr}_1, \dots, \mathbf{tr}_n$, and
default functions $\mathbf{dv}_1, \dots, \mathbf{dv}_n$,

is admissible under the history \mathbf{h} provided

- (a) **const** is a new function symbol of \mathbf{n} arguments,
(**base** is a new function symbol of no arguments,
if a base function is supplied), **r**, $\mathbf{ac}_1, \dots, \mathbf{ac}_n$
are new function symbols of one argument, and
all the above function symbols are distinct;
- (b) each \mathbf{tr}_i is a type restriction over the recognizers
of \mathbf{h} together with the symbol **r**;
- (c) for each \mathbf{i} , \mathbf{dv}_i is either **base** or one of the
base functions of \mathbf{h} ; and

- (d) for each i , if dv_i is **base** then r satisfies tr_i
and otherwise the type of dv_i satisfies tr_i .

If the tr_i are not specified, they should each be assumed to be **(NONE-OF)**.

If admissible, the act adds the axioms shown below. In the special case that no **base** is supplied, **T** should be used for all occurrences of $(r \text{ (base)})$ below, and **F** should be used for all terms of the form $(EQUAL \ x \text{ (base)})$ below.

- (1) $(OR \ (EQUAL \ (r \ X) \ T) \ (EQUAL \ (r \ X) \ F))$
- (2) $(r \ (const \ X1 \ \dots \ Xn))$
- (3) $(r \ (base))$
- (4) $(NOT \ (EQUAL \ (const \ X1 \ \dots \ Xn) \ (base)))$
- (5) $(IMPLIES \ (AND \ (r \ X) \ (NOT \ (EQUAL \ X \ (base)))) \ (EQUAL \ (const \ (ac_1 \ X) \ \dots \ (ac_n \ X)) \ X))$

For each i from 1 to n , the following formula

- (6) $(IMPLIES \ trt_i \ (EQUAL \ (ac_i \ (const \ X1 \ \dots \ Xn)) \ Xi))$

where trt_i is the type restriction term for tr_i and Xi .

For each i from 1 to n , the following formula

- (7) $(IMPLIES \ (OR \ (NOT \ (r \ X)) \ (OR \ (EQUAL \ X \ (base)) \ (AND \ (NOT \ trt_i) \ (EQUAL \ X \ (const \ X1 \ \dots \ Xn))))) \ (EQUAL \ (ac_i \ X) \ (dv_i)))$

where trt_i is the type restriction term for tr_i and Xi .

For each recognizer, r' , in the recognizer functions of h the formula

- (8) $(IMPLIES \ (r \ X) \ (NOT \ (r' \ X)))$

- (9) (**IMPLIES** (**r** **X**)
 (**EQUAL** (**COUNT** **X**)
 (**IF** (**EQUAL** **X** (**base**))
 (**ZERO**)
 (**ADD1**
 (**PLUS**
 (**COUNT** (**ac**₁ **X**))
 ...
 (**COUNT** (**ac**_n **X**))) \otimes (**ZERO**))))))

- (10) The **SUBRP** axiom for each of the symbols **const**, **base** (if supplied), **r**, **ac**₁, ..., **ac**_n. We define the “**SUBRP** axiom” on page 174.

Note. In the first edition, the shell principle included two additional axioms, there labeled (8) and (9), which were in fact derivable from axiom (10) of the first edition. Their deletion from the second edition has caused renumbering of the subsequent axioms.

4.5.3. Natural Numbers—Axioms 12.*n*

Shell Definition.

Add the shell **ADD1** of one argument
 with base function **ZERO**,
 recognizer function **NUMBERP**,
 accessor function **SUB1**,
 type restriction (**ONE-OF** **NUMBERP**), and
 default function **ZERO**.

Note. In Appendix II we explicitly list the axioms added by this invocation of the shell principle. Each axiom has a number of the form **12.*n***, where *n* indicates the corresponding axiom schema of the shell principle.

Axiom 13.

(**NUMBERP** (**COUNT** **X**))

Axiom 14.

(**EQUAL** (**COUNT** **T**) (**ZERO**))

Axiom 15.

(**EQUAL** (**COUNT** **F**) (**ZERO**))

Defining Axiom 16.

```
(ZEROP X)
=
(OR (EQUAL X (ZERO)) (NOT (NUMBERP X)))
```

Defining Axiom 17.

```
(FIX X) = (IF (NUMBERP X) X (ZERO))
```

Defining Axiom 18.

```
(PLUS X Y)
=
(IF (ZEROP X)
    (FIX Y)
    (ADD1 (PLUS (SUB1 X) Y)))
```

4.5.4. Ordered Pairs—Axioms 19.*n***Shell Definition.**

Add the shell **CONS** of two arguments with recognizer function **LISTP**, accessor functions **CAR** and **CDR**, and default functions **ZERO** and **ZERO**.

Notes. This invocation of the shell principle is, strictly speaking, inadmissible because there are axioms about **CONS**, **CAR**, and **CDR** in the **SUBRP** axioms added on behalf of the preceding shell. We ignore this inadmissibility and add the corresponding axioms anyway.

In Appendix II we explicitly list the axioms added by this invocation of the shell principle. Each axiom has a number of the form **19.*n***, where *n* indicates the corresponding axiom schema of the shell principle.

4.5.5. Literal Atoms—Axioms 20.*n***Shell Definition.**

Add the shell **PACK** of one argument with recognizer function **LITATOM**, accessor function **UNPACK**, and default function **ZERO**.

Notes. This invocation of the shell principle is, strictly speaking, inadmissible because there are axioms about **PACK** in the **SUBRP** axioms added on behalf of the preceding shells. We add the corresponding axioms anyway.

In Appendix II we explicitly list the axioms added by this invocation of the shell principle. Each axiom has a number of the form **20.n**, where n indicates the corresponding axiom schema of the shell principle.

4.5.6. *Negative Integers*—Axioms **21.n**

Shell Definition.

Add the shell **MINUS** of one argument
with recognizer function **NEGATIVEP**,
accessor function **NEGATIVE-GUTS**,
type restriction **(ONE-OF NUMBERP)**, and
default function **ZERO**.

Note. In Appendix II we list explicitly the axioms added by this invocation of the shell principle. Each axiom has a number of the form **21.n**, where n indicates the corresponding axiom schema of the shell principle.

4.6. Explicit Value Terms

Note. This section is technically an aside in the development of the logic. We define a particularly important class of terms in the logic, called the “explicit value terms.” Intuitively, the explicit value terms are the “canonical constants” in the logic. It is almost the case that every constant term—every variable-free term—can be mechanically reduced to a unique, equivalent explicit value. The only terms not so reducible are those involving (at some level in the definitional hierarchy) undefined functions, constrained functions, or calls of metafunctions such as **V&C\$**. Thus, the explicit value terms are the terms upon which we can “compute” in the logic. They are the basis for our encoding of the terms as objects in the logic, and elaborate syntactic conventions are adopted in the extended syntax to permit their succinct expression.

Terminology. We say **tr** is the i^{th} *type restriction* for a constructor function symbol **fn** of arity n iff $1 \leq i \leq n$, and **tr** is the i^{th} type restriction specified in the axiomatic act in which **fn** was introduced.

Examples. The first type restriction for **ADD1** is **(ONE-OF NUMBERP)**. The second type restriction for **CONS** is **(NONE-OF)**.

Terminology. We say **t** is an *explicit value term* in a history h iff **t** is a term and either (a) **t** is a call of a base function symbol in h , or (b) **t** is a call of a

constructor function symbol **fn** in *h* on arguments **a**₁, ..., **a**_{*n*} and for each $1 \leq i \leq n$, **a**_{*i*} is an explicit value term in *h* and the type of the top function symbol of **a**_{*i*} satisfies the *i*th type restriction for the constructor function **fn**. We frequently omit reference to the history *h* when it is obvious by context.

Examples. The following are explicit value terms:

```
(ADD1 (ADD1 (ZERO)))
```

```
(CONS (PACK (ZERO)) (CONS (TRUE) (ADD1 (ZERO))))
```

The term **(ADD1 X)** is not an explicit value, since **X** is neither a call of a base function symbol nor a call of a constructor. The term **(ADD1 (TRUE))** is not an explicit value, because the top function symbol of **(TRUE)** does not satisfy the type restriction, **(ONE-OF NUMBERP)**, for the first argument of **ADD1**.

4.7. The Extended Syntax

Note on the Second Edition. The second edition of this handbook extends the syntax of the first edition by adding

- the Common Lisp convention for writing comments both with semi-colon and with balanced occurrences of **#|** and **|#**;
- more of Common Lisp's notation for writing integers, including binary, octal and hexadecimal notation, e.g., **#B1000** as an abbreviation of 8;
- Common Lisp's "backquote" notation, so that **`(A ,@X B)** is an abbreviation of **(CONS 'A (APPEND X (CONS 'B NIL)))**;
- **COND** and **CASE**, which permit the succinct abbreviations of commonly used nests of **IF**-expressions;
- **LIST***, which permits the succinct abbreviation of commonly used nests of **CONS**-expressions; and
- **LET**, which permits the "binding" of "local variables" to values, permitting the succinct abbreviation of terms involving multiple occurrences of identical subterms.

To explain the new integer notation and the backquote notation in a way that is (we believe) perfectly accurate and permits their use "inside **QUOTES**" it was necessary to redevelop the foundation of the syntax as it was presented in the first edition. In particular, in the second edition we start with a class of structures larger than the first edition's "s-expressions"—structures which include

such utterances as $\backslash (, \mathbf{X})$. Because the foundation changed, virtually all of the first edition's section 4.7 (pages 112-124), has changed. But the new syntax is strictly an extension of the old; every well-formed term in the old "extended syntax" is well formed in the new and abbreviates the same formal term. So despite the relatively massive change to the description, the impact of the second edition is only to add and fully document the features noted above.

In the first edition, Appendix I contained a formalization of the old syntax and abbreviation conventions; however, that formalization was developed independently of the informal description of the syntax (though the two were carefully scrutinized for consistency). In this edition we have undertaken in Appendix I to formalize not just the syntax and abbreviation conventions but this particular informal description of them. Thus, virtually every concept defined informally in this section (and in section 4.11.2) is defined formally in Appendix I. Readers who are struggling with the problem of formalizing some system—and here we do not mean to limit ourselves just to formal languages—might benefit from a comparative study of this section and Appendix I. Here we describe in the mathematician's precise English a collection of concepts that we there render formally.

Finally, to aid the new user in trying to understand our abbreviation conventions, we have included the machine readable text of Appendix I among our standard example files, as file 10. By noting this library one can execute the formal concepts. Comments in the file give some tips for making experimentation convenient.

Notes. The extended syntax differs from the formal syntax only in that it permits certain abbreviations. That is, every term in the formal syntax is also a term in the extended syntax, but the extended syntax admits additional well-formed utterances that are understood to stand for certain formal terms. These abbreviations primarily concern notation for shell constants such as numbers, literal atoms, and lists. In addition, the extended syntax provides some abbreviations for commonly used function nests and for the general purpose bounded quantifier function **FOR**. We delay the presentation of the quantifier abbreviations until after we have axiomatized **FOR** (see section 4.11, page 177) but discuss all other aspects of the extended syntax in this section.

We define the extended syntax in six steps.

1. We define a set of tree structures called "token trees" that includes the formal terms and some other structures as well.
2. We explain how token trees are displayed.
3. We identify three nested subsets of the token trees: the "readable" token trees, the "s-expressions," and the "well-formed" s-expressions.

4. We define a mapping, called “readmacro expansion,” from the readable token trees to s-expressions.
5. We define a mapping, called “translation,” from the well-formed s-expressions to formal terms.
6. Finally, we make the convention that a readable token tree may be used as a term in the extended syntax provided its readmacro expansion is well formed. Such a token tree abbreviates the translation of its readmacro expansion.

For example, the token tree we display as `(LIST* X #B010 '(0 . 1))` is readable. Its readmacro expansion is `(LIST* X 2 (QUOTE (0 . 1)))`. This s-expression is well formed. Its translation is the formal term:

```
(CONS X
  (CONS (ADD1 (ADD1 (ZERO)))
    (CONS (ZERO)
      (ADD1 (ZERO))))).
```

Thus, `(LIST* X #B010 '(0 . 1))` may be used as a term in the extended syntax and abbreviates the `CONS`-nest above.

The token tree `(LIST* X ,A)` is not readable, because it violates rules on the use of the “the comma escape from backquote.” The token tree `(EQUAL '(,X . ,Y))` is readable. However, its readmacro expansion is `(EQUAL (CONS X Y))`, which is not well formed, because the function symbol `EQUAL` requires two arguments.

We apologize to readers expecting a definition of our syntax presented in a formal grammar (e.g., BNF). We have three reasons for proceeding in this fashion. First, despite the apparent simplicity of our syntax, it has extremely powerful and complicated provisions for describing structures. These provisions allow a natural embedding of the language into its constants, which facilitates the definition of a formal metatheory in the logic, as carried out in the final sections of this chapter. Thus, much of the verbiage here devoted to syntax can be thought of as devoted to the formal development of the metatheory.

Second, we not only wish to specify the legal expressions in the extended syntax but also to map them to terms in the formal syntax. We think it unlikely that an accurate formal presentation of our syntax and its meaning would be any more clear than the informal but precise one offered here; furthermore, it would be much less accessible to most readers.

Finally, this presentation is closely related to the actual implementation of the syntax in the Nqthm system. In our implementation the user types “displayed token trees.” These character strings are read by the Common Lisp `read` routine. The `read` routine causes an error if the token tree presented is not “readable” (e.g., uses a comma outside a backquote). Otherwise, the `read`

routine “expands” the “**read** macros” (single quote, backquote, and #) and returns the s-expression as a Lisp object. It is only then that our theorem prover gets to inspect the object to determine if it is “well formed” and, if so, what term it abbreviates.

In Appendix I we not only formalize the concepts defined below but we provide a recursive descent parser that recognizes when a string of ASCII character codes is the display of a token tree.

4.7.1. Token Trees and Their Display

Note. Our token trees are essentially the parse trees of Common Lisp s-expressions, except for the treatment of **read** macro characters (such as # and ') and certain atoms. For example, **#B010**, **+2.** and **2** are three *distinct* token trees. We define “readmacro expansion” so that these three trees expand into the same “s-expression.” We start the development with the definition of the # convention for writing down integers in various bases. Then we introduce the tokens involved in the single quote and backquote conventions. Finally, we define token trees and how to display them.

Terminology. A character *c* is a *base n digit character* if and only if $n \leq 16$ and *c* is among the first *n* characters in the sequence **0123456789ABCDEF**. The position of *c* (using zero-based indexing) in the sequence is called its *digit value*.

Note. When we define how we display “token trees” and the digit characters in them we will make it clear that case is irrelevant. Thus, while this definition makes it appear that only the upper case characters **A-F** are digit characters, we will effectively treat **a-f** as digit characters also.

Example. The base 2 digits are **0** and **1**. Their digit values are 0 and 1, respectively. Among the base 16 digits are **A** and **F**. The digit value of **A** is 10 and the digit value of **F** is 15.

Terminology. A sequence of characters, *s*, is a *base n digit sequence* if and only if *s* is nonempty and every element of *s* is a base *n* digit character. The *base n value* of a base *n* digit sequence $c_k \dots c_1 c_0$ is the integer $c_k n^k + \dots + c_1 n^1 + c_0 n^0$, where c_i is the value of the digit c_i .

Example. **1011** is a base 2 digit sequence. Its base 2 value is the integer eleven. **1011** is also a base 8 digit sequence. Its base 8 value is the integer five hundred and twenty one.

Terminology. A sequence of characters, s , is an *optionally signed base n digit sequence* if and only if s is either a base n digit sequence or s is nonempty, the first character of s is either a $+$ or $-$, and the remaining characters constitute a base n digit sequence. If the first character of s is $-$, the *base n signed value* of s is the negative of the base n value of the constituent base n digit sequence. Otherwise, the *base n signed value* of s is the base n value of the constituent base n digit sequence.

Example. **A2** and **+A2** are both optionally signed base 16 digit sequences whose signed values (in decimal notation) are both 162. **-A2** is also such a sequence; its signed value is -162.

Terminology. A sequence of characters, s , is a *#-sequence* if and only if the length of s is at least three, the first character in s is $\#$, the second character in s is in the set $\{\mathbf{B} \ \mathbf{O} \ \mathbf{X}\}$, and the remaining characters in s constitute an optionally signed base n digit sequence, where n is 2, 8, or 16 according to whether the second character of s is **B**, **O**, or **X**, respectively.

Note. Our convention of disregarding case will allow **b**, **o**, and **x** in the second character position of #-sequences.

Terminology. Finally, a sequence of characters, s , is a *numeric sequence* if and only if one of the following is true:

- s is an optionally signed base 10 digit sequence (in which case its *numeric value* is the base 10 signed value of s);
- s is an optionally signed base 10 digit sequence with a dot character appended on the right (in which case its *numeric value* is the base 10 signed value of s with the dot removed);
- s is a #-sequence (in which case its *numeric value* is the base n signed value of the sequence obtained from s by deleting the first two characters, where n is 2, 8, or 16 according to whether the second character of s is **B**, **O**, or **X**).

Example. Table 4.3 shows some numeric sequences and their numeric values written in decimal notation.

Terminology. The character sequence containing just the single quote (sometimes called “single gritch”) character ($'$) is called the *single quote token*. The character sequence containing just the backquote character ($`$) is called the *backquote token*. The character sequence containing just the dot character ($.$) is called the *dot token*. The character sequence containing just the comma

Table 4.3

sequence	value
123	123
-5.	-5
#B1011	11
#O-123	-83
#xfab	4011

character (,) is called the *comma token*. The character sequence containing just the comma character followed by the at-sign character (,@) is called the *comma at-sign token*. The character sequence containing just the comma character followed by the dot character (,.) is called the *comma dot token*.

Terminology. A sequence of characters, s , is a *word* if and only if s is a numeric sequence or s is nonempty and each character in s is a member of the set

{A B C D E F G H I J K L M
 N O P Q R S T U V W X Y Z
 0 1 2 3 4 5 6 7 8 9
 \$ ^ & * _ - + = ~ { } ? < > }.

Note. All of our symbols are words, but the set of words is larger because it includes such non-symbols as ***1*TRUE**, **123**, **1AF**, and **#B1011** that begin with non-alphabetic characters.

Terminology. A *token tree* is one of the following:

- an integer;
- a word;
- a nonempty sequence of token trees (a token tree of this kind is said to be an *undotted* token tree);
- a sequence of length at least three whose second-to-last element is the dot token and all of whose other elements are token trees (a token tree of this kind is said to be a *dotted* token tree); or
- a sequence of length two whose first element is one of the tokens specified below and whose second element is a token tree (called the *constituent token tree*):
 - the single quote token (such a token tree is said to be a *single quote* token tree),

- the backquote token (a *backquote* token tree),
- the comma token (a *comma escape* token tree), or
- the comma at-sign or comma dot tokens (such token trees are said to be *splice escape* token trees).

Note. In order to explain how token trees are displayed we must first introduce the notion of “comments.” Following Common Lisp, we permit both “semi-colon comments” and “number sign comments.” The latter are, roughly speaking, text delimited by *balanced* occurrences of `#|` and `|#`. To define this notion precisely, we must introduce the terminology to let us talk about `#|` and `|#` clearly.

Terminology. The integer i is a *#|-occurrence* in the character string s of length n iff $0 \leq i \leq n-2$, the i^{th} (0-based) character of s is the number sign character (`#`) and the $i+1^{\text{st}}$ character of s is the vertical bar character, `|`. We define what it is for i to be a *#|-occurrence* in strict analogy. A *|#-occurrence*, j , is *strictly to the right* of a *#|-occurrence*, i , iff $j > i+1$. Finally, if i is a *#|-occurrence* in s and j is a *|#-occurrence* in s that is strictly to the right, then the *substring of s delimited by i and j* is the substring of s that begins with the $i+2^{\text{nd}}$ character and ends with the $j-1^{\text{st}}$ character.

Examples. Consider the string `#|Comment|`. The string has eleven characters in it. The only *#|-occurrence* is 0. The only *|#-occurrence* is 9. The substring delimited by these occurrences is `Comment`. In the string `#||#` the only *#|-occurrence* is 0 and the only *|#-occurrence* is 2, which is strictly to the right of the former. The substring delimited by these two occurrences is the empty string.

Terminology. A string of characters, s , has *balanced number signs* iff all the *#|-* and *|#-* occurrences can be paired (i.e., put into 1:1 correspondence) so that every *#|* has its corresponding *|#* strictly to its right and the text delimited by the paired occurrences has balanced number signs.

Examples. The string `Comment` has balanced number signs because there are no *#|-* or *|#-* occurrences. The string

`code #|Comment|` and `code`

also has balanced number signs. The following string does not have balanced number signs:

`#|code #|Comment|` and `code`.

Terminology. A string of characters, s , is a *#-comment* iff s is obtained from a string, s' , that has balanced number signs, by adding a number sign character immediately followed by a vertical bar ($\#|$) to the left-hand end and a vertical bar immediately followed by a number sign ($| \#$) to the right-hand end.

Notes. Generally speaking, any string of characters not including $\#|$ or $| \#$ can be made into a *#-comment* by surrounding the string with $\#|$ and $| \#$. Such comments will be allowed in certain positions in the display of terms. Roughly speaking, the recognition that a string is a comment is license to ignore the string; that is the whole point of comments. Comments can be added to the display of a term without changing the term displayed. We are more specific in a moment. The display of such a commented term can itself be turned into a *#-comment* by again surrounding it with $\#|$ and $| \#$. If we did not insist on “balancing number signs” the attempt to “comment out” a term could produce ill-formed results because the “opening” $\#|$ would be “closed” by the first $| \#$ encountered in the term’s comments and the rest of the term would not be part of the comment.

The question of where *#-comments* are allowed is difficult to answer, but it is answered below. The problem is that in Common Lisp (whose **read** routine we actually use to parse terms) the number sign character does not automatically terminate the parsing of a lexeme. Thus, $(\mathbf{H} \#| \mathbf{text} | \# \mathbf{X})$ reads as a list containing the lexeme \mathbf{H} followed by the lexeme \mathbf{X} , but $(\mathbf{H} \#| \mathbf{text} | \# \mathbf{X})$ reads as the list containing the lexeme $\mathbf{H} \# \mathbf{text} \#$ followed by the lexeme \mathbf{X} . So some “white space” must appear after \mathbf{H} and before the number sign in order for the number sign to be read as the beginning of a comment. As for the end of the comment, the $| \#$ closes the comment no matter what follows. Thus, $(\mathbf{H} \#| \mathbf{text} | \# \mathbf{X})$ is also read as \mathbf{H} followed by \mathbf{X} , even though there is no space after the second number sign. The naive reader might conclude that $\#|$ must always be preceded by white space and $| \#$ may optionally be followed by white space. Unfortunately, the rule is a little more complicated. The display $(\mathbf{FN} (\mathbf{H} \mathbf{X}) \#| \mathbf{text} | \# \mathbf{X})$ is read as $(\mathbf{FN} (\mathbf{H} \mathbf{X}) \mathbf{X})$. The reason is that the close parenthesis in $(\mathbf{H} \mathbf{X})$ is a “lexeme terminating character” that terminates the lexeme before it, i.e., \mathbf{X} , and constitutes a lexeme in itself. Thus, the $\#|$ is recognized as beginning a comment. We now explain this precisely.

Terminology. The *white space characters* are defined to be space, tab, and newline (i.e., carriage return). The *lexeme terminating characters* are defined to be semicolon, open parenthesis, close parenthesis, single quote mark, backquote mark, and comma. (Note that $\#$ is *not* a lexeme terminator!) The union of the white space characters and the lexeme terminating characters is called the *break characters*.

Terminology. A *comment* is any of the following:

- a (possibly empty) sequence of white space characters,
- a **#**-comment,
- a sequence of characters that starts with a semicolon, ends with a newline, and contains no other newlines, or
- the concatenation of two comments.

A comment is said to be a *separating comment* if it is nonempty and its first character is either a white space character or a semicolon.

Example. Thus,

```
;This is a comment.
```

is a separating comment. The string **#|And this is a comment.|#** is a comment but not a separating comment. It can be made into a separating comment by adding a space before the first **#**. The following is a display of a separating comment (since it starts with a semicolon) that illustrates that comments may be strung together:

```
; This is a long comment.  
; It spans several lines and  
#|Contains both semicolon and  
number sign comments. In addition,  
immediately after the following  
number sign is a white space  
comment.|# ; And here's another  
;semicolon comment.
```

Terminology. Suppose s_1 and s_2 are nonempty character strings. Then *suitable separation* between s_1 and s_2 is any comment with the following properties. Let c_1 be the last character in s_1 , and let c_2 be the first character in s_2 .

- If c_1 is a break character, suitable separation is any comment.
- If c_1 is not a break character but c_2 is a break character, suitable separation is either the empty comment or any separating comment.
- If neither c_1 nor c_2 is a break character, suitable separation is any separating comment.

When we use the phrase “suitable separation” it is always in a context in which s_1 and s_2 are implicit.

Terminology. A *suitable trailer* after a string **s** is any suitable separation between **s** and a string beginning with a break character. That is, a suitable trailer may be any comment if the last character of **s** is a break character and otherwise may be either an empty comment or a separating comment.

Examples. We use suitable separations and trailers to separate the lexemes in our displays of terms in the extended syntax. For example, in **(FN X)** the lexeme **FN** is separated from the lexeme **X** by the separating comment consisting of a single space. Any separating comment is allowed in the extended syntax. Thus,

```
(FN;Comment
 X)
```

is an acceptable display. The empty string is not a separating comment. Thus, **(FNX)** is an unacceptable display. Similarly, because **#|Comment|#** is not a separating comment, **(FN#|Comment|#X)** is unacceptable. On the other hand, by adding a space to the front of the **#**-comment above we obtain a separating comment and thus **(FN #|Comment|#X)** is an acceptable display. Now consider **(FN (H X) Y)**. Because the last character of **FN** is not a break character but open parenthesis is a break character, suitable separation between displays of **FN** and **(H X)** is either the empty string or a separating comment. Thus, the first four of the following five displays will be acceptable. The last will not, because the comment used is not a separating comment or the empty comment.

```
(FN(H X) Y)           ; the empty comment

(FN
 (H X) Y)             ; a white space comment

(FN;text
 (H X) Y)             ; a semicolon comment

(FN #|text|# (H X) Y) ; a separating #-comment

(FN#|text|# (H X) Y)  ; unacceptable!
```

Any comment may be used to separate **(H X)** and **Y**, because the close parenthesis character is a break character. Thus, **(FN(H X)Y)** will be an allowed display, as will **(FN(H X)#|text|#Y)**.

Terminology. To *display* a token tree that is an integer or word, we write down a comment, a decimal representation of the integer or the characters in the word, followed by a suitable trailer. Upper case characters in a word may be displayed

in lower case. To *display* a dotted or undotted token tree consisting of the elements t_1, \dots, t_n , where t_{n-1} may or may not be the dot token, we write down a comment, an open parenthesis, suitable separation, a display of t_1 , suitable separation, a display of t_2 , suitable separation, ..., a display of t_n , suitable separation, a close parenthesis, and a comment, where we display the dot token as though it were a word. All other token trees consist of a token and a constituent token tree. To display them we write down a comment, the characters in the token, a comment, a display of the constituent token tree, and a suitable trailer.

Note. What token tree displays as **123**? Unfortunately, there are two: the tree consisting of the integer 123 and the tree consisting of the numeric sequence **123**. These two trees readmacro expand to the same tree and since readmacro expansion is involved in our abbreviation convention, it is irrelevant which of the two trees was actually displayed.

Example. Below we show some (displays of) token trees:

```
(EQUAL X                               ;Comments may
      (QUOTE (A B . C))) ;be included

(EQUAL X (QUOTE (A B . C)))

(equal x (quote (a b . c)))

(QUOTE (A . B))

'(A . B)

`(A ,X ,@(STRIP-CARS Y) B)

((1AB B**3 C) 12R45 (ADD1))
```

The first, second and third are different displays of the same token tree—the only difference between them is the comments used and the case of the characters. The fourth and fifth are displays of two different token trees, both of length two with the same second element, **(A . B)**, but one having the word **QUOTE** as its first element and the other having the single quote token as its first element. Note the difference in the way they are displayed. It will turn out that these two different token trees readmacro expand to the same s-expression. The sixth display shows how a backquote token tree containing two escapes may be displayed. Once we have defined readmacro expansion and translation it will be clear that all of the token trees except the last abbreviate terms.

Notes. Because of our convention in this book of using lower case typewriter font words as metavariables, we will refrain from using lower case when displaying token trees. Thus, if **fn** is understood to be any function symbol of two arguments, then **(PLUS X Y)** is to be understood to be “of the form” of the token tree we display as **(fn X Y)** while it is not “of the form” of the tree we display as **(FN X Y)**. The reader may wonder why we permit token trees to be displayed in lower case if we never so display them. The point is that we never so display them in this book. But in other papers in which formulas are displayed, users frequently find lowercase characters more attractive. Furthermore, our implementation of Nqthm inherits from the host Lisp system the default action of raising lower case characters to upper case characters when reading from the user’s terminal or files. (Special action has to be taken to prevent this.) Thus, many users type formulas in lower case. Since metavariables are not part of the system, this presents no ambiguity.

In Appendix I we define the inverse of the display process: a parser that determines whether a list of ASCII character codes is a display of a token tree and, if so, what token tree was displayed (up to the integer/numeric sequence ambiguity which is resolved by always adopting the numeric sequence interpretation). The parser is defined in two passes, one that breaks the list of character codes into lexemes and a second which attempts to construct a token tree from the lexemes.

4.7.2. *Readable Token Trees, S-Expressions and Readmacro Expansion*

Note. The token tree displayed as **`(, ,X)** is problematic because it contains a doubly-nested backquote escape in a singly nested backquote tree. The token trees displayed as **` ,@X** and **`(1 . ,@X)** are problematic because they use the **,@** notation in “non-element positions.” We make precise these requirements by defining the notion of “readable” token trees for depth *n* and then restricting our attention later to the readable token trees for depth 0.

Terminology. A token tree **s** is *readable* from depth *n* iff one of the following holds:

- **s** is an integer or word;
- **s** is an undotted token tree and each element of **s** is readable from *n*;

- **s** is a dotted token tree and each element of **s** (other than the dot token) is readable from *n* and the last element of **s** is not a splice escape tree;
- **s** is a single quote token tree and the constituent token tree is readable from *n*;
- **s** is a backquote token tree and the constituent token tree is readable from *n*+1 and is not a splice escape tree; or
- **s** is a comma escape or splice escape token tree, *n*>0, and the constituent token tree is readable from *n*-1.

Example. The token tree (**A ,X B**) is not readable from depth 0 but is readable from depth 1. Thus, **`(A ,X B)** is readable from depth 0. The expressions **`,`@X** and (**A . ,@X**) are not readable from any depth.

Terminology. An *s-expression* is a token tree containing no numeric sequences or tokens other than the dot token.

Terminology. If **s** is a token tree readable from depth 0, then its *readmacro expansion* is the *s-expression* obtained by the following four successive transformations.

- Replace every numeric sequence by its numeric value.
- Replace every single quote token by the word **QUOTE**.
- Replace every backquote token tree, **`x**, by the “backquote expansion” of **x** (see below), replacing innermost trees first.
- At this point the transformed tree contains no tokens other than, possibly, the dot token. Replace every subtree of **s** of the form, (**x₁ x₂ ... x_k . (y₁ ... y_n)**), by (**x₁ x₂ ... x_k y₁ ... y_n**) and every subtree of the form (**x₁ x₂ ... x_k . NIL**) by (**x₁ x₂ ... x_k**). That is, a subtree should be replaced if it is a dotted token tree and its last element is either a dotted or undotted token tree or the word **NIL**. Let **x** be such a tree and let **y** be its last element. If **y** is a dotted or undotted token tree, then **x** is replaced by the concatenation of **y** onto the right-hand end of the sequence obtained from **x** by deleting the dot and the last element. If **y** is the word **NIL** then **x** is replaced by the sequence obtained from **x** by deleting the dot and the last element.

Examples. The readmacro expansion of (**A B . ' (#B100 . C)**) proceeds in the following steps. First, the numeric sequence is replaced by its integer value, producing (**A B . ' (4 . C)**). Second, the single quote token tree is

replaced by a **QUOTE** tree, **(A B . (QUOTE (4 . C)))**. Since there are no backquote tokens in the tree, that step of readmacro expansion is irrelevant here. Finally we consider the dotted token trees. The innermost one is not of the form that is replaced. The outermost one is replaceable. The result is **(A B QUOTE (4 . C))**.

The readmacro expansion of **(PLUS #B100 (SUM '(1 2 3)))** is **(PLUS 4 (SUM (QUOTE (1 2 3))))**. Note that the ambiguity concerning the display of integers and numeric sequences makes it impossible to uniquely determine the token tree initially displayed. For example, we do not know if it contained the integer two or the numeric sequence 2. However, the second tree displayed is the readmacro expansion of the first. As such, it is an s-expression and contains no numeric sequences. Thus, the second token tree is uniquely determined by the display and the fact that it is known to be an s-expression.

Note. We now define “backquote expansion.” It is via this process that **`(,X ,Y . ,Z)** is readmacro expanded into **(CONS X (CONS Y Z))**, for example. Our interpretation of backquote is consistent with the Common Lisp interpretation [131, 132]. However, Common Lisp does not specify what s-expression is built by backquote; instead it specifies what the value of the s-expression must be. Different implementations of Common Lisp actually produce different readmacro expansions. For example, in one Common Lisp, **`(,X)** might be **(CONS X NIL)** and in another it might be **(LIST X)**. The values of these expressions are the same. But consider what happens when a backquoted form is quoted, e.g., **``(,X)**. In one of the two hypothetical Lisps mentioned above, this string reads as the constant **'(CONS X NIL)**, while in the other it reads as **'(LIST X)**. This is intolerable in our setting since it would mean that the token tree **(EQUAL (CAR ``(,X)) 'CONS)** might read as a theorem in one Common Lisp and read as a non-theorem in another. We therefore have to choose a particular readmacro expansion of backquote (consistent with Common Lisp). We document it here. It is implemented by suitable changes to the Lisp readtable in Nqthm.⁹

Terminology. If **s** is a token tree readable from some depth **n** and **s** contains no numeric sequences, single quote tokens or backquote tokens, then its *backquote expansion* is the token tree defined as follows.

- If **s** is an integer or a word, then **(QUOTE s)** is its backquote expansion.

⁹Actually, we do not force our implementation of backquote onto the Nqthm user typing Lisp at the top-level. See **BACKQUOTE-SETTING** in the Reference Guide, page 301.

- If **s** is a comma escape token tree, **,x**, or a splice escape token tree, **,@x** or **,.x**, then **x** is its backquote expansion.
- If **s** is an undotted token tree or a dotted token tree, then its backquote expansion is the “backquote-list expansion” of **s** (see below).

Terminology. We now define the *backquote-list expansion* for a dotted or undotted token tree, **s**, that is readable from some depth *n* and contains no numeric sequences, single quote tokens, or backquote tokens. Let **x** be the backquote expansion of the first element of **s**. Let **y** be as defined by cases below. Then the backquote-list expansion of **s** is either **(APPEND x y)** or **(CONS x y)**, according to whether the first element of **s** is a splice escape token tree or not. The definition of **y** is by cases:

- If **s** is a token tree of length 1, then **y** is **(QUOTE NIL)**.
- If **s** is a dotted token tree of length 3, then **y** is the backquote expansion of the last element of **s**.
- If **s** is any other dotted or undotted token tree, then **y** is the backquote-list expansion of the token tree obtained by removing the first element from **s**.

Examples. To illustrate the first case above, where **s** is an undotted token tree of length 1, consider the backquote-list expansion of **(A)**. The result is **(CONS (QUOTE A) (QUOTE NIL))** because the backquote expansion of **A** is **(QUOTE A)**. The second case is illustrated by **(B . ,Z)**. Its backquote-list expansion is **(CONS (QUOTE B) Z)**. We combine these to illustrate the third case. The backquote-list expansion of **((A) B . ,Z)** is **(CONS (CONS (QUOTE A) (QUOTE NIL)) (CONS (QUOTE B) Z))**. The backquote-list expansion of **(A ,@Z B)** is **(CONS (QUOTE A) (APPEND Z (CONS (QUOTE B) (QUOTE NIL))))**.

Since the readmacro expansion of **`(A)** is just the backquote expansion of **(A)**, which, in turn, is the backquote-list expansion of **(A)**, we see that the readmacro expansion of **`(A)** is **(CONS (QUOTE A) (QUOTE NIL))**. In Table 4.4 we show some other examples of the readmacro expansion of backquote trees.

We can explain the last example of Table 4.4. The readmacro expansion of **``(, ,X)** proceeds by first replacing the innermost backquote tree by its backquote expansion. So let us consider the backquote expansion of **`(, ,X)**. We might first observe that while this tree is not readable from depth 0, its backquote expansion is nevertheless well defined. Indeed, its backquote expansion is the token tree **(CONS ,X (QUOTE NIL))** because the backquote expansion

Table 4.4

token tree	readmacro expansion
<code>`(X ,X ,@X)</code>	<code>(CONS (QUOTE X) (CONS X (APPEND X (QUOTE NIL))))</code>
<code>`((A . ,X) (B . ,Y) . ,REST)</code>	<code>(CONS (CONS (QUOTE A) X) (CONS (CONS (QUOTE B) Y) REST))</code>
<code>``(, ,X)</code>	<code>(CONS (QUOTE CONS) (CONS X (CONS (CONS (QUOTE QUOTE) (CONS (QUOTE NIL) (QUOTE NIL))) (QUOTE NIL))))</code>

of `, ,X` is `,X`. So replacing the constituent token tree, ``(, ,X)`, of ```(, ,X)` by its backquote expansion produces ``(CONS ,X (QUOTE NIL))`. It is easy to check that the readmacro expansion of this tree is as shown in the table.

Observe that backquote expansion does not always yield an s-expression: the backquote expansion of `, ,X` is `,X`. However, the backquote expansion of a token tree readable from depth 0 produces an s-expression because each escape token tree is embedded in enough backquotes to ensure that all the escape and backquote tokens are eliminated.

4.7.3. Some Preliminary Terminology

Note. We have described how readmacro expansion transforms readable token trees into s-expressions. We now turn to the identification of a subset of the s-expressions, called “well-formed” s-expressions and the formal terms they abbreviate. We need a few preliminary concepts first.

Terminology. The *NUMBERP* corresponding to a nonnegative integer n is the term `(ZERO)` if n is 0, and otherwise is the term `(ADD1 t)`, where t is the *NUMBERP* corresponding to $n-1$. The *NEGATIVEP* corresponding to a negative integer n is the term `(MINUS t)`, where t is the *NUMBERP* corresponding to $-n$.

Examples. The **NUMBERP** corresponding to 2 is (**ADD1** (**ADD1** (**ZERO**))). The **NEGATIVEP** corresponding to -1 is (**MINUS** (**ADD1** (**ZERO**))).

Terminology. If **fn** is a **CAR/CDR** symbol, we call the sequence of characters in **fn** starting with the second and concluding with the next to last the *A/D sequence* of **fn**.

Terminology. If **s** is a character sequence of **A**s and **D**s, the *CAR/CDR nest* for **s** around a term **b** is the term **t** defined as follows. If **s** is empty, **t** is **b**. Otherwise, **s** consists of either an **A** or **D** followed by a sequence **s'**. Let **t'** be the **CAR/CDR** nest for **s'** around **b**. Then **t** is (**CAR t'**) or (**CDR t'**), according to whether the first character of **s** is **A** or **D**.

Example. The symbol **CADDAAR** is a **CAR/CDR** symbol. Its **A/D** sequence is the sequence **ADDAA**. The **CAR/CDR** nest for **ADDAA** around **L** is (**CAR** (**CDR** (**CDR** (**CAR** (**CAR L**))))).

Terminology. We say a term **e** is the *explosion* of a sequence of ASCII characters, **s**, iff either (a) **s** is empty and **e** is (**ZERO**) or (b) **s** is a character **c** followed by some sequence **s'** and **e** is (**CONS i e'**) where **i** is the **NUMBERP** corresponding to the ASCII code for **c** and **e'** is the explosion of **s'**.

Note. See the definition of **ASCII-TABLE** in Appendix I for the codes of the printing ASCII characters.

Example. The ASCII codes for the characters **A**, **B**, and **C** are 65, 66, and 67 respectively. Let **t₆₅**, **t₆₆**, and **t₆₇** denote, respectively, the **NUMBERPs** corresponding to 65, 66, and 67. For example, **t₆₅** here denotes a nest of **ADD1**s 65 deep with a (**ZERO**) at the bottom. Then the explosion of **ABC** is the formal term

$$(\text{CONS } t_{65} (\text{CONS } t_{66} (\text{CONS } t_{67} (\text{ZERO}))))).$$

Terminology. We say the term **e** is the *LITATOM* corresponding to a symbol **s** iff **e** is the term (**PACK e'**) where **e'** is the explosion of **s**.

Example. The **LITATOM** corresponding to the symbol **ABC** is

$$(\text{PACK} (\text{CONS } t_{65} (\text{CONS } t_{66} (\text{CONS } t_{67} (\text{ZERO}))))),$$

where **t₆₅**, **t₆₆**, and **t₆₇** are as in the last example.

4.7.4. Well-Formed S-Expressions and Their Translations

Notes. The terminology we have developed is sufficient for defining what it means for an s-expression to be “well formed” and what its “translation” is, for all s-expressions except those employing **QUOTE** or abbreviated **FORs**. Rather than define the concepts necessary to pin down these conventions, we now jump ahead in our development of the syntax and define “well formed” and “translation.” Such a presentation here necessarily involves undefined concepts—the notions of well-formedness and translation of both **QUOTE** and **FOR** expressions. However, by providing the definition at this point in the development we can use some s-expressions to illustrate and motivate the discussion of the more elaborate notations.

Terminology. Below we define two concepts: what it means for an s-expression \mathbf{x} to be a *well-formed term in the extended syntax* and, for well-formed s-expressions, what is the *translation* into a term in the formal syntax. These definitions are made implicitly with respect to a history because **QUOTE** notation permits the abbreviation of explicit values, a concept which, recall, is sensitive to the history. Our style of definition is to consider any s-expression \mathbf{x} and announce whether it is well formed or not and if so, what its translation is.

- If \mathbf{x} is an integer, it is well formed and its translation is the explicit value term denoted by \mathbf{x} (see page 158).
- If \mathbf{x} is a symbol then
 - If \mathbf{x} is **T**, it is well formed and its translation is the formal term (**TRUE**).
 - If \mathbf{x} is **F**, it is well formed and its translation is the formal term (**FALSE**).
 - If \mathbf{x} is **NIL**, it is well formed and its translation is the explicit value term denoted by **NIL** (see page 158).
 - If \mathbf{x} is any other symbol, it is well formed and its translation is the formal term \mathbf{x} .
- If \mathbf{x} is a dotted s-expression, it is not well formed.
- If the first element of \mathbf{x} is the symbol **QUOTE**, then \mathbf{x} is well formed iff it is of the form (**QUOTE** \mathbf{e}) where \mathbf{e} is an explicit value descriptor (see page 158). If well formed, the translation of \mathbf{x} is the explicit value term denoted by \mathbf{e} (see page 158).
- If the first element of \mathbf{x} is the symbol **COND**, then the well-formedness of \mathbf{x} and its translation are defined by cases as follows.

- If \mathbf{x} is of the form $(\text{COND } (\mathbf{T} \ \mathbf{v}))$, then it is well formed iff \mathbf{v} is well formed. If \mathbf{x} is well formed, the translation of \mathbf{x} is the translation of \mathbf{v} .
 - If \mathbf{x} is of the form $(\text{COND } (\mathbf{w} \ \mathbf{v}) \ \mathbf{x}_1 \ \dots \ \mathbf{x}_n)$, where $n > 0$, then \mathbf{x} is well formed iff \mathbf{w} is not \mathbf{T} and \mathbf{w} , \mathbf{v} and $(\text{COND } \mathbf{x}_1 \ \dots \ \mathbf{x}_n)$ are well formed. If \mathbf{x} is well formed, let **test**, **val**, and **rest** be, respectively, the translations of \mathbf{w} , \mathbf{v} , and $(\text{COND } \mathbf{x}_1 \ \dots \ \mathbf{x}_n)$. Then the translation of \mathbf{x} is $(\text{IF test val rest})$.
 - Otherwise, \mathbf{x} is not well formed.
- If the first element of \mathbf{x} is the symbol **CASE**, then the well-formedness of \mathbf{x} and its translation are defined by cases as follows.
 - If \mathbf{x} is of the form $(\text{CASE } \mathbf{w} \ (\text{OTHERWISE } \mathbf{v}))$, then \mathbf{x} is well formed iff \mathbf{w} and \mathbf{v} are well formed. If \mathbf{x} is well formed, the translation of \mathbf{x} is the translation of \mathbf{v} .
 - If \mathbf{x} is of the form $(\text{CASE } \mathbf{w} \ (\mathbf{e} \ \mathbf{v}) \ \mathbf{x}_1 \ \dots \ \mathbf{x}_n)$, where $n > 0$, then \mathbf{x} is well formed iff no \mathbf{x}_i is of the form $(\mathbf{e} \ \mathbf{x}'_i)$ and in addition \mathbf{w} , $(\text{QUOTE } \mathbf{e})$, \mathbf{v} , and $(\text{CASE } \mathbf{w} \ \mathbf{x}_1 \ \dots \ \mathbf{x}_n)$ are well formed. If \mathbf{x} is well formed, then let **key**, **obj**, **val**, and **rest** be, respectively, the translations of \mathbf{w} , $(\text{QUOTE } \mathbf{e})$, \mathbf{v} , and $(\text{CASE } \mathbf{w} \ \mathbf{x}_1 \ \dots \ \mathbf{x}_n)$. Then the translation of \mathbf{x} is $(\text{IF } (\text{EQUAL key obj}) \ \text{val rest})$.
 - Otherwise, \mathbf{x} is not well formed.
 - If the first element of \mathbf{x} is the symbol **LET**, then \mathbf{x} is well formed iff \mathbf{x} is of the form $(\text{LET } ((\mathbf{w}_1 \ \mathbf{v}_1) \ \dots \ (\mathbf{w}_n \ \mathbf{v}_n)) \ \mathbf{y})$, the \mathbf{w}_i , \mathbf{v}_i and \mathbf{y} are well formed, the translation of each \mathbf{w}_i is a symbol and the translation of \mathbf{w}_i is different from that of \mathbf{w}_j when i is different from j . If \mathbf{x} is well formed, let \mathbf{var}_i be the translation of \mathbf{w}_i , let \mathbf{t}_i be the translation of \mathbf{v}_i and let **body** be the translation of \mathbf{y} . Let \mathbf{s} be the substitution that replaces \mathbf{var}_i by \mathbf{t}_i , i.e., $\{\langle \mathbf{var}_1, \mathbf{t}_1 \rangle \dots \langle \mathbf{var}_n, \mathbf{t}_n \rangle\}$. Then the translation of \mathbf{x} is **body/s**.
 - Otherwise, \mathbf{x} is of the form $(\text{fn } \mathbf{x}_1 \ \dots \ \mathbf{x}_n)$, where **fn** is a symbol other than **QUOTE**, **COND**, **CASE**, or **LET**, and each \mathbf{x}_i is an s-expression. If some \mathbf{x}_i is not well formed, then \mathbf{x} is not well formed. Otherwise, let \mathbf{t}_i be the translation of \mathbf{x}_i below:
 - If **fn** is in the set $\{\text{NIL } \mathbf{T} \ \mathbf{F}\}$, \mathbf{x} is not well formed.

- If **fn** is the symbol **LIST** then **x** is well formed and its translation is the **CONS** nest around the translation of **NIL** for t_1, \dots, t_n .
- If **fn** is the symbol **LIST*** then **x** is well formed iff $n \geq 1$. If **x** is well formed, its translation is the **CONS** nest around t_n for t_1, \dots, t_{n-1} .
- If **fn** is a **CAR/CDR** symbol, then **x** is well formed iff n is 1 and, if well formed, its translation is the **CAR/CDR** nest around t_1 for the **A/D** sequence of **fn**.
- If **fn** is a function symbol of arity n , then **x** is well formed and its translation is the formal term $(fn\ t_1\ \dots\ t_n)$.
- If **fn** is the symbol **FOR** and n is 5 or 7, then **x** is well formed iff **x** is an abbreviated **FOR** (see page 178). If well formed, the translation of **x** is the **FOR** expression denoted by **x** (see page 178).
- If **fn** is in the set **{AND OR PLUS TIMES}** and $n > 2$, then **x** is well formed and its translation is the **fn** nest around t_n for t_1, \dots, t_{n-1} .
- Otherwise, **x** is not well formed.

Examples. Table 4.5 shows well-formed s-expressions on the left and their translations to formal terms on the right.

Table 4.5

s-expression	translation
T	(TRUE)
2	(ADD1 (ADD1 (ZERO)))
(COND (P X) (Q Y) (T Z))	(IF P X (IF Q Y Z))
(LIST* A B C)	(CONS A (CONS B C))
(CADR X)	(CAR (CDR X))
(PLUS I J K)	(PLUS I (PLUS J K))

Certain formal terms, such as the translation of **NIL**, are exceedingly painful to write down because they contain deep nests of **ADD1**s. Table 4.6 also contains translations, except this time the right-hand column is, technically, not a formal term but rather another well-formed s-expression with the same translation. In particular, in Table 4.6 we use decimal notation in the right-hand column, but otherwise confine ourselves to formal syntax.

Table 4.6

s-expression	s-expression with same translation
NIL	(PACK (CONS 78 (CONS 73 (CONS 76 0))))
(LIST 1 2 3)	(CONS 1 ;first element (CONS 2 ;second element (CONS 3 ;third element (PACK ;NIL (CONS 78 (CONS 73 (CONS 76 0))))))

4.7.5. QUOTE Notation

Notes and Example. In this subsection we define what we mean by an “explicit value descriptor” and the “explicit value denoted” by such a descriptor. These are the concepts used to define the well-formedness and meaning of s-expressions of the form (**QUOTE e**).

Each explicit value term can be written in **QUOTE** notation. That is, for each explicit value term **t** there is exactly one s-expression **e** such that the s-expression (**QUOTE e**) is well formed and translates to **t**. We call **e** the “explicit value descriptor” of **t**. For example, consider the s-expression

```
(CONS 1
  (CONS (PACK
    (CONS 65 (CONS 66 (CONS 67 0))))
    (CONS 2 3))).
```

This s-expression is well formed and translates to an explicit value— indeed,

except for the use of decimal notation, this s-expression is an explicit value. Call that explicit value term \mathbf{t} . The explicit value descriptor for \mathbf{t} is the s-expression $(\mathbf{1\ ABC\ 2\ .\ 3})$. Thus, the translation of $(\mathbf{QUOTE\ (1\ ABC\ 2\ .\ 3)})$ is \mathbf{t} .

Our **QUOTE** notation is derived from the Lisp notation for data structures composed of numbers, symbols, and ordered pairs, but is complicated by the need to denote structures containing user-defined shell constants. That is, after the theory has been extended by the addition of a new shell, it is possible to build constants containing both primitive shells and user-defined ones, e.g., lists of stacks. Unlike Lisp's **QUOTE** notation, the notation described here permits such constants to be written down, via an "escape" mechanism.

Following the precedent set for well-formedness and translation, we proceed in a top-down fashion to define what we mean by an "explicit value descriptor" and its "denoted explicit value" without first defining the terminology to discuss the "escape" mechanism. Immediately following the definition below we illustrate the use of **QUOTE** notation on primitive shell constants, e.g., lists, numbers, and literal atoms. We define the escape mechanism for user-declared shells in the next subsection.

Terminology. Below we define two concepts: what it is for an s-expression \mathbf{e} to be an *explicit value descriptor* and, for explicit value descriptors, what is the *denoted explicit value term*. These definitions are made with respect to a history which is used implicitly below. Our style of definition is to consider any s-expression \mathbf{e} and announce whether it is an explicit value descriptor or not and if so, what its denoted explicit value term is.

- If \mathbf{e} is an integer, it is an explicit value descriptor and the explicit value term it denotes is the **NEGATIVEP** or **NUMBERP** corresponding to \mathbf{e} , according to whether \mathbf{e} is negative or not.
- If \mathbf{e} is a word, then
 - If \mathbf{e} is the word ***1*TRUE**, \mathbf{e} is an explicit value descriptor and denotes the explicit value **(TRUE)**.
 - If \mathbf{e} is the word ***1*FALSE**, \mathbf{e} is an explicit value descriptor and denotes the explicit value **(FALSE)**.
 - If \mathbf{e} is a symbol, \mathbf{e} is an explicit value descriptor and denotes the **LITATOM** corresponding to \mathbf{e} .
 - Otherwise, \mathbf{e} is not an explicit value descriptor.
- If the first element of \mathbf{e} is the word ***1*QUOTE**, \mathbf{e} is an explicit value descriptor iff it is an explicit value escape descriptor (see the next subsection). If so, it has the form **(*1*QUOTE fn e₁ ...**

e_n), where fn is a constructor or base function symbol of arity n and each e_i is an explicit value descriptor denoting an explicit value t_i . The explicit value denoted by e is then $(fn\ t_1\ \dots\ t_n)$. (That this is, indeed, an explicit value is assured by the definition of “explicit value escape descriptor.”)

- If e is a dotted s-expression of length 3, i.e., $(e_1\ .\ e_2)$, then e is an explicit value descriptor iff each e_i is an explicit value descriptor. If so, let t_i be the explicit value denoted by e_i . Then the explicit value denoted by e is $(CONS\ t_1\ t_2)$.
- If e is an s-expression of length 1, i.e., (e_1) , e is an explicit value descriptor iff e_1 is an explicit value descriptor. If so, let t_1 be the explicit value denoted by e_1 and let nil be the explicit value denoted by **NIL**. Then the explicit value denoted by e is $(CONS\ t_1\ nil)$.
- Otherwise, either e is a dotted s-expression of length greater than 3 or e is a non-dotted s-expression of length greater than 1. Let e_1 be the first element of e and let e_2 be the sequence consisting of the remaining elements of e . Observe that e_1 and e_2 are both s-expressions. e is an explicit value descriptor iff each e_i is an explicit value descriptor. If so, let t_i be the explicit value denoted by e_i . Then the explicit value denoted by e is $(CONS\ t_1\ t_2)$.

Examples. Table 4.7 illustrates the **QUOTE** notation. The two columns contain token trees rather than s-expressions simply to save space—we write $'x$ in place of the s-expression $(QUOTE\ x)$. Each token tree is readable and read-macro expands to a well-formed s-expression. The s-expressions of the left-hand column are all examples of **QUOTE** forms. The s-expressions of the right-hand column use **QUOTE** only to represent literal atoms. Corresponding s-expressions in the two columns have identical translations.

Note. Of particular note is the possible confusion of the meaning of the symbol **T** (and, symmetrically, of **F**) in s-expressions. If **T** is used “outside a **QUOTE**” it denotes **(TRUE)**. If **T** is used “inside a **QUOTE**” it denotes the literal atom whose “print name” is the single character **T**. To include **(TRUE)** among the elements of a **QUOTEd** list, the non-symbol ***1*TRUE** should be written.

If the s-expression $(QUOTE\ ABC)$ is used as a term it denotes the term also denoted by $(PACK\ (QUOTE\ (65\ 66\ 67\ .\ 0)))$. However, if the s-expression $(QUOTE\ ABC)$ is used “inside a **QUOTE**,” i.e., as an explicit value descriptor as in the term $(QUOTE\ (QUOTE\ ABC))$, it denotes the term also denoted by $(LIST\ (QUOTE\ QUOTE)\ (QUOTE\ ABC))$. The translation of

Table 4.7

token tree for s-expression in QUOTE notation	token tree for s-expression with same translation
'123	123
'ABC	(PACK '(65 66 67 . 0))
'(65 66 67 . 0)	(CONS 65 (CONS 66 (CONS 67 0)))
'(PLUS I J)	(CONS 'PLUS (CONS 'I (CONS 'J 'NIL)))
'((I . 2) (J . 3))	(LIST (CONS 'I 2) (CONS 'J 3))
'((A . *1*TRUE) (B . T))	(LIST (CONS 'A (TRUE)) (CONS 'B (PACK (CONS 84 0))))

(QUOTE ABC) is a **LITATOM** constant; the translation of (QUOTE (QUOTE ABC)) is a **LISTP** constant.

Here is another example illustrating the subtlety of the situation. The innocent reader may have, by now, adopted the convention that whenever (CADR X) is seen, (CAR (CDR X)) is used in its place. This is incorrect. When (CADR X) is used as a term, i.e., when we are interested in its translation into a formal term, it denotes (CAR (CDR X)). But if (CADR X) is “inside a QUOTE” it is not being used as a term but rather as an explicit value descriptor. In particular, the translation of (QUOTE (CADR X)) is a list whose first element is the **LITATOM** denoted by the term (QUOTE CADR), not a list whose first element is the **LITATOM** denoted by (QUOTE CAR). While the translations of (CADR X) and (CAR (CDR X)) are the same, the translations of (QUOTE (CADR X)) and (QUOTE (CAR (CDR X))) are different. Similarly the translation of (QUOTE 1) is not the same as that of (QUOTE (ADD1 (ZERO))); the first is a **NUMBERP**, the second is a **LISTP**.

4.7.6. **1*QUOTE Escape Mechanism for User Shells*

Notes and Example. In this section we describe how to use the ***1*QUOTE** convention to write down user-declared shell constants. In particular, we define the notion of the “explicit value escape descriptor” used above.

Roughly speaking, an explicit value escape descriptor is an s-expression of the form **(*1*QUOTE fn e₁ ... e_n)** where **fn** is a shell constructor or base function and the **e_i** are explicit value descriptors denoting its arguments. Thus, if **PUSH** is a shell constructor of two arguments and **EMPTY** is the corresponding base function then **(*1*QUOTE PUSH 3 (*1*QUOTE EMPTY))** is an explicit value escape descriptor, and hence an explicit value descriptor, that denotes the constant term also denoted by **(PUSH 3 (EMPTY))**. We restrict the legal escape descriptors so that the mechanism cannot be used to write down alternative representations of constants that can be written in the conventional **QUOTE** notation. For example, **(*1*QUOTE CONS 1 2)** is *not* an explicit value escape descriptor because if it were it would be an alternative representation of **(CONS 1 2)**. Furthermore, we must restrict the escape descriptors so that they indeed denote explicit values. Is **(PUSH 3 (EMPTY))** an explicit value? The answer depends upon the type restrictions for the **PUSH** shell. To answer this question it is necessary to know the current history.

Terminology. The s-expression **e** is an *explicit value escape descriptor* with respect to a history **h** iff **e** has the form **(*1*QUOTE fn e₁ ... e_n)** and each of the following is true:

- **fn** is a constructor or base function symbol of arity **n** in history **h**;
- **fn** is not **ADD1**, **ZERO**, or **CONS**;
- each **e_i** is an explicit value descriptor with corresponding denoted term **t_i** in **h**;
- if **fn** is a constructor, the top function symbol of each **t_i** satisfies the corresponding type restriction for **fn**;
- if **fn** is **PACK**, **t₁** is not the explosion of any symbol; and
- if **fn** is **MINUS**, **t₁** is **(ZERO)**.

Notes and Examples. We now illustrate the use of the ***1*QUOTE** escape mechanism. Suppose we are in a history obtained by extending the current one with the following:

Shell Definition.

Add the shell **PUSH** of 2 arguments
 with base function **EMPTY**,
 recognizer function **STACKP**,
 accessor functions **TOP** and **POP**,
 type restrictions **(ONE-OF NUMBERP)** and **(ONE-OF STACKP)**, and
 default functions **ZERO** and **EMPTY**.

Table 4.8 contains some example s-expressions employing the ***1*QUOTE** mechanism. To save space we again exhibit token trees whose readmacro expansions are the s-expressions in question.

Table 4.8

token tree for s-expression	token tree for s-expression with same translation
'(A (*1*QUOTE MINUS 0))	(LIST 'A (MINUS 0))
'(((*1*QUOTE PUSH 2 (*1*QUOTE PUSH 3 (*1*QUOTE EMPTY))) FOO . 45)	(CONS (PUSH 2 (PUSH 3 (EMPTY))) (CONS 'FOO 45))
'(*1*QUOTE PACK (97 98 99 . 0))	(PACK '(97 98 99 . 0))

***1*QUOTE** can be used not only to denote constants of “new” types but also to write down “unusual” constants of the primitive types, namely **(MINUS 0)** and “**LITATOMs** corresponding to non-symbols.”

***1*QUOTE** notation is actually a direct reflection of the internal representation of shell constants in the Nqthm system. If **STACKP** constants, say, are allowed as terms, then it is desirable to have a unique representation of them that can be used in the representation of other constants as well. The representation we developed is the one suggested by ***1*QUOTE** notation. We did not have to make this implementation decision be visible to the user of the logic. We could have arranged for only the primitive data types to be abbreviated by **QUOTE** notation and all other constants built by the application of constructor and base functions. Making the convention visible is actually an expression of our opinion that the syntax should not hide too much from the user. For example, the user writing and verifying metafunctions will appreciate knowing the internal forms.

Finally, it should be noted that backquote notation can often be used where `*1*QUOTE` is otherwise needed. For example, `(QUOTE (A (*1*QUOTE EMPTY) B))` can also be written as ``(A ,(EMPTY) B)`. The translation of the former is the same as the translation of the readmacro expansion of the latter.

4.7.7. The Definition of the Extended Syntax

Abbreviation. When a token tree that is readable from depth 0 and that is not an s-expression is used as an s-expression, we mean the s-expression obtained by readmacro expanding the token tree. Thus, henceforth, when we say something like “consider the s-expression ‘`ABC`’” we mean “consider the s-expression `(QUOTE ABC)`.”

Terminology. The *extended syntax* consists of the token trees readable from depth 0 whose readmacro expansions are well formed.

Abbreviation. When an expression in the extended syntax is used as a term, it is an abbreviation for the translation of its readmacro expansion. When an expression in the extended syntax is used as a formula, it is an abbreviation for `t≠(FALSE)`, where `t` is the translation of the readmacro expansion of the expression.

Note. In Appendix I we exhibit a parser for the extended syntax as a function defined within the logic.

4.8. Ordinals

Defining Axiom 22.

```
(LESSP X Y)
=
(IF (ZEROP Y)
  F
  (IF (ZEROP X)
    T
    (LESSP (SUB1 X) (SUB1 Y))))
```

Axiom 23.

```
(IMPLIES (NOT (ZEROP I))
  (LESSP (SUB1 I) I))
```

Note. **Axiom 23** permits us to apply the induction principle to prove the fundamental properties of **LESSP**, **PLUS**, and **COUNT**, which in turn permit us to induct in more sophisticated ways.

Defining Axiom 24.

```
(ORD-LESSP X Y)
=
(IF (NOT (LISTP X))
    (IF (NOT (LISTP Y))
        (LESSP X Y)
        T)
    (IF (NOT (LISTP Y))
        F
        (IF (ORD-LESSP (CAR X) (CAR Y))
            T
            (AND (EQUAL (CAR X) (CAR Y))
                 (ORD-LESSP (CDR X) (CDR Y)))))))
```

Defining Axiom 25.

```
(ORDINALP X)
=
(IF (LISTP X)
    (AND (ORDINALP (CAR X))
         (NOT (EQUAL (CAR X) 0))
         (ORDINALP (CDR X))
         (OR (NOT (LISTP (CDR X)))
              (NOT (ORD-LESSP (CAR X) (CADR X)))))
    (NUMBERP X))
```

Examples. See page 49 in the primer for a table of example ordinals.

4.9. Useful Function Definitions

4.9.1. Boolean Equivalence

Defining Axiom 26.

```
(IFF P Q)
=
(IF P
    (IF Q T F)
    (IF Q F T))
```


4.9.2. *Natural Number Arithmetic*

Defining Axiom 27.

```
(GREATERP I J) = (LESSP J I)
```

Defining Axiom 28.

```
(LEQ I J) = (NOT (LESSP J I))
```

Defining Axiom 29.

```
(GEQ I J) = (NOT (LESSP I J))
```

Defining Axiom 30.

```
(MAX I J) = (IF (LESSP I J) J (FIX I))
```

Defining Axiom 31.

```
(DIFFERENCE I J)
=
(IF (ZEROP I)
    0
    (IF (ZEROP J)
        I
        (DIFFERENCE (SUB1 I) (SUB1 J)))))
```

Defining Axiom 32.

```
(TIMES I J)
=
(IF (ZEROP I)
    0
    (PLUS J (TIMES (SUB1 I) J)))
```

Defining Axiom 33.

```
(QUOTIENT I J)
=
(IF (ZEROP J)
    0
    (IF (LESSP I J)
        0
        (ADD1 (QUOTIENT (DIFFERENCE I J) J)))))
```

Defining Axiom 34.

```
(REMAINDER I J)
=
(IF (ZEROP J)
    (FIX I)
    (IF (LESSP I J)
        (FIX I)
        (REMAINDER (DIFFERENCE I J) J)))
```

4.9.3. List Processing**Defining Axiom 35.**`(NLISTP X) = (NOT (LISTP X))`**Defining Axiom 35a.**`(IDENTITY X) = X`**Defining Axiom 36.**`(APPEND L1 L2)``=``(IF (LISTP L1)
 (CONS (CAR L1) (APPEND (CDR L1) L2))
 L2)`**Defining Axiom 37.**`(MEMBER X L)``=``(IF (NLISTP L)
 F
 (IF (EQUAL X (CAR L))
 T
 (MEMBER X (CDR L))))`**Defining Axiom 38.**`(UNION L1 L2)``=``(IF (LISTP L1)
 (IF (MEMBER (CAR L1) L2)
 (UNION (CDR L1) L2)
 (CONS (CAR L1) (UNION (CDR L1) L2)))
 L2)`**Defining Axiom 39.**`(ADD-TO-SET X L)``=``(IF (MEMBER X L)
 L
 (CONS X L))`

Defining Axiom 40.

```
(ASSOC X ALIST)
=
(IF (NLISTP ALIST)
    F
    (IF (EQUAL X (CAAR ALIST))
        (CAR ALIST)
        (ASSOC X (CDR ALIST)))))
```

Defining Axiom 41.

```
(PAIRLIST L1 L2)
=
(IF (LISTP L1)
    (CONS (CONS (CAR L1) (CAR L2))
          (PAIRLIST (CDR L1) (CDR L2)))
    NIL)
```

4.10. The Formal Metatheory

Note. In this section we describe the interpreter for the logic. We start by presenting the notion of the “quotation” of terms. Roughly speaking, the quotation of a term is an explicit value that has a structure isomorphic to that of the term; for example, the quotation of **(PLUS X Y)** is the explicit value **'(PLUS X Y)**. An important property of quotations is that, for most terms, the interpreted value of the quotation under a certain standard assignment is equal to the term. For example, the value of **'(PLUS X Y)** as determined by our interpreter, when **'X** has the value **X** and **'Y** has the value **Y**, is **(PLUS X Y)**. After defining quotations we define the interpreter. Finally, we describe the **SUBRP** and non-**SUBRP** axioms that tie **QUOTE**d symbols to the interpreter.

4.10.1. The Quotation of Terms

Note. The “quotation” of an explicit value term may be rendered either by nests of constructor function applications or by embedding the term in a **QUOTE** form. This makes the notion of “quotation” depend upon the notion of “explicit value,” which, recall, involves a particular history *h* from which the constructor and base functions are drawn. This is the only sense in which the notion of “quotation” depends upon a history.

Terminology. We say **e** is a *quotation* of **t** (in some history *h* which is implicit throughout this definition) iff **e** and **t** are terms and either (a) **t** is a variable

symbol and **e** is the **LITATOM** corresponding to **t**, (b) **t** is an explicit value term and **e** is (**LIST 'QUOTE t**), or (c) **t** has the form (**fn a₁ ... a_n**) and **e** is (**CONS efn elst**) where **efn** is the **LITATOM** corresponding to **fn** and **elst** is a “quotation list” (see below) of **a₁ ... a_n**. Note that clauses (b) and (c) are not mutually exclusive.

Terminology. We say **elst** is a *quotation list of tlst* (in some history *h* which is implicit throughout this definition) iff **elst** is a term and **tlst** is a sequence of terms, and either (a) **tlst** is empty and **elst** is **NIL** or (b) **tlst** consists of a term **t** followed by a sequence **tlst'** and **elst** is (**CONS e elst'**) where **e** is a quotation of **t** and **elst'** is a quotation list of **tlst'**.

Examples. In Table 4.9 we give some terms and examples of their quotations.

Table 4.9

term	quotation displayed in the extended syntax
ABC	'ABC
(ZERO)	'(ZERO)
(ZERO)	'(QUOTE 0)
(PLUS 3 (TIMES X Y))	'(PLUS (QUOTE 3) (TIMES X Y))

Note. To describe the axioms for the **BODY** function, we wish to say something like “for each defined function symbol, **fn**, (**BODY 'fn**) is the quotation of the body of the definition of **fn**.” But note that explicit values, e.g., **(ZERO)** above, have multiple quotations. (Indeed, all terms containing explicit values have multiple quotations.) Consequently, we cannot speak of “the” quotation of a term. To get around this we define the notion of the “preferred quotation.” The preferred quotation of **(ZERO)** is **'(QUOTE 0)**. In general, the definitions of “preferred quotation” and “preferred quotation list,” below, are strictly analogous to the definitions of “quotation” and “quotation list,” above, except that explicit values must be encoded in **'QUOTE** form. This is done by making clauses (b) and (c) of the definition of “quotation” be mutually exclusive with clause (b) the superior one.

Terminology. We say **e** is the *preferred quotation* of **t** (in some history **h** which is implicit throughout this definition) iff **e** and **t** are terms and either (a) **t** is a variable symbol and **e** is the **LITATOM** corresponding to **t**, (b) **t** is an explicit value term and **e** is (**LIST 'QUOTE t**), or (c) **t** has the form (**fn a₁ ... a_n**), **t** is not an explicit value, and **e** is (**CONS efn elst**) where **efn** is the **LITATOM** corresponding to **fn** and **elst** is the “preferred quotation list” (see below) of **a₁ ... a_n**.

Terminology. We say **elst** is the *preferred quotation list of tlst* (in some history **h** which is implicit throughout this definition) iff **elst** is a term and **tlst** is a sequence of terms, and either (a) **tlst** is empty and **elst** is **NIL** or (b) **tlst** consists of a term **t** followed by a sequence **tlst'** and **elst** is (**CONS e elst'**) where **e** is the preferred quotation of **t** and **elst'** is the preferred quotation list of **tlst'**.

4.10.2. V&C\$ and EVAL\$

Note. The axiomatization of **V&C\$** and **EVAL\$** are rather subtle. In the primer, starting on page 52, we explain many of the constraints and “design decisions.” In addition, the interested reader is urged to see [30].

Defining Axiom 42.

```
(FIX-COST VC N)
=
(IF VC
  (CONS (CAR VC) (PLUS N (CDR VC)))
  F)
```

Defining Axiom 43.

```
(STRIP-CARS L)
=
(IF (NLISTP L)
  NIL
  (CONS (CAAR L) (STRIP-CARS (CDR L))))
```

Defining Axiom 44.

```
(SUM-CDRS L)
=
(IF (NLISTP L)
  0
  (PLUS (CDAR L) (SUM-CDRS (CDR L))))
```

Note. We now “define” **V&C\$**. This axiom defines a partial function and would not be admissible under the definitional principle. Because of its complexity we include comments in the axiom.

Defining Axiom 45.

(V&C\$ FLG X VA)

=

(IF (EQUAL FLG 'LIST)

 ;X is a list of terms. Return a list of value-cost
 ;“pairs”—some “pairs” may be F.

 (IF (NLISTP X)
 NIL
 (CONS (V&C\$ T (CAR X) VA)
 (V&C\$ 'LIST (CDR X) VA)))

 ;Otherwise, consider the cases on the X.

(IF (LITATOM X) ;Variable
 (CONS (CDR (ASSOC X VA)) 0)

(IF (NLISTP X) ;Constant
 (CONS X 0)

(IF (EQUAL (CAR X) 'QUOTE) ;QUOTEd
 (CONS (CADR X) 0)

(IF (EQUAL (CAR X) 'IF) ;IF-expr

 ;If the test of the IF is defined, test the value and
 ;interpret the appropriate branch. Then, if the branch
 ;is defined, increment its cost by that of the test plus
 ;one. If the test is undefined, X is undefined.

 (IF (V&C\$ T (CADR X) VA)
 (FIX-COST
 (IF (CAR (V&C\$ T (CADR X) VA))
 (V&C\$ T (CADDR X) VA)
 (V&C\$ T (CADDRR X) VA))
 (ADD1 (CDR (V&C\$ T (CADR X) VA))))
 F)

;Otherwise, **X** is the application of a **SUBRP** or
 ;defined function. If some argument is undefined, so is **X**.

```
(IF (MEMBER F (V&C$ 'LIST (CDR X) VA))
    F
```

```
(IF (SUBRP (CAR X))                ;SUBRP
```

;Apply the primitive to the values of the arguments and
 ;let the cost be one plus the sum of the argument costs.

```
(CONS (APPLY-SUBR (CAR X)
                 (STRIP-CARS
                  (V&C$ 'LIST (CDR X) VA)))
      (ADD1 (SUM-CDRS
              (V&C$ 'LIST (CDR X) VA))))
```

```
;Defined fn
```

;Interpret the **BODY** on the values of the arguments
 ;and if that is defined increment the cost by one plus
 ;the sum of the argument costs.

```
(FIX-COST
 (V&C$ T (BODY (CAR X))
 (PAIRLIST
  (FORMALS (CAR X))
  (STRIP-CARS (V&C$ 'LIST (CDR X) VA))))
 (ADD1
  (SUM-CDRS
   (V&C$ 'LIST (CDR X) VA)))))))))
```

Note. Having defined **V&C\$** we can now define the general purpose “apply” function in terms of it:

Defining Axiom 46.

```
(V&C-APPLY$ FN ARGS)
=
(IF (EQUAL FN 'IF)
  (IF (CAR ARGS)
    (FIX-COST (IF (CAAR ARGS)
                  (CADR ARGS)
                  (CADDR ARGS))
              (ADD1 (CDAR ARGS)))
    F)
  (IF (MEMBER F ARGS)
    F
    (IF (SUBRP FN)
      (CONS (APPLY-SUBR
              FN
              (STRIP-CARS ARGS))
            (ADD1 (SUM-CDRS ARGS)))
      (FIX-COST
        (V&C$ T
          (BODY FN)
          (PAIRLIST (FORMALS FN)
                    (STRIP-CARS ARGS)))
        (ADD1 (SUM-CDRS ARGS)))))))
```

Note. A trivial consequence of the definitions of **V&C\$** and **V&C-APPLY\$** is that the following is a theorem:

Theorem.

```
(V&C$ FLG X VA)
=
(IF (EQUAL FLG 'LIST)
  (IF (NLISTP X)
    NIL
    (CONS (V&C$ T (CAR X) VA)
          (V&C$ 'LIST (CDR X) VA)))
  (IF (LITATOM X) (CONS (CDR (ASSOC X VA)) 0)
    (IF (NLISTP X) (CONS X 0)
      (IF (EQUAL (CAR X) 'QUOTE)
        (CONS (CADR X) 0)
        (V&C-APPLY$
          (CAR X)
          (V&C$ 'LIST (CDR X) VA)))))))
```


Note. We finally define the functions **APPLY\$** and **EVAL\$**:

Defining Axiom 47.

```
(APPLY$ FN ARGS)
=
(CAR (V&C-APPLY$ FN (PAIRLIST ARGS 0)))
```

Defining Axiom 48.

```
(EVAL$ FLG X A)
=
(IF (EQUAL FLG 'LIST)
  (IF (NLISTP X)
    NIL
    (CONS (EVAL$ T (CAR X) A)
          (EVAL$ 'LIST (CDR X) A)))
  (IF (LITATOM X) (CDR (ASSOC X A))
    (IF (NLISTP X) X
      (IF (EQUAL (CAR X) 'QUOTE) (CADR X)
        (APPLY$ (CAR X)
                  (EVAL$ 'LIST (CDR X) A)))))))
```

4.10.3. The SUBRP and non-SUBRP Axioms

Notes. We now axiomatize the functions **SUBRP**, **APPLY-SUBR**, **FORMALS**, and **BODY** and define what we mean by the “SUBRP” and “non-SUBRP axioms.”

The function **SUBRP** is Boolean:

Axiom 49.

```
(OR (EQUAL (SUBRP FN) T) (EQUAL (SUBRP FN) F))
```

The three functions **SUBRP**, **FORMALS**, and **BODY** “expect” **LITATOMs** as arguments, i.e., the quotations of function symbols. We tie down the three functions outside their “expected” domain with the following three axioms:

Axiom 50.

```
(IMPLIES (NOT (LITATOM FN)) (EQUAL (SUBRP FN) F))
```

Axiom 51.

```
(IMPLIES (NOT (LITATOM FN)) (EQUAL (FORMALS FN) F))
```

Axiom 52.

```
(IMPLIES (NOT (LITATOM FN)) (EQUAL (BODY FN) F))
```

Note. In a similar spirit, we define the **FORMALS** and **BODY** of **SUBRPs** to be **F**, and we define the result of applying a non-**SUBRP** with **APPLY-SUBR** to be **F**:

Axiom 53.

(IMPLIES (SUBRP FN) (EQUAL (FORMALS FN) F))

Axiom 54.

(IMPLIES (SUBRP FN) (EQUAL (BODY FN) F))

Axiom 55.

(IMPLIES (NOT (SUBRP FN)) (EQUAL (APPLY-SUBR FN X) F))

Note. In section 4.12 we enumerate the primitive **SUBRPs** and non-**SUBRPs**. For each we will add either the “**SUBRP** axiom” or the “non-**SUBRP** axiom,” which we now proceed to define.

Terminology. We say term **t** is the **n**th *CDR* nest around the term **x** iff **n** is a natural number and either (a) **n** is 0 and **t** is **x** or (b) **n**>0 and **t** is (CDR **t'**) where **t'** is the **n**-1st *CDR* nest around **x**. When we write (CDRⁿ **x**) where a term is expected it is an abbreviation for the **n**th *CDR* nest around **x**.

Example. (CDR² **A**) is (CDR (CDR **A**)).

Terminology. The *SUBRP* axiom for **fn**, where **fn** is a function symbol of arity **n**, is

(AND (EQUAL (SUBRP 'fn) T)
 (EQUAL (APPLY-SUBR 'fn L)
 (fn (CAR (CDR⁰ L))
 ...
 (CAR (CDRⁿ⁻¹ L))))))

where 'fn is the **LITATOM** corresponding to **fn**.

Example. The *SUBRP* axiom for **PLUS** is

(AND (EQUAL (SUBRP 'PLUS) T)
 (EQUAL (APPLY-SUBR 'PLUS L)
 (PLUS (CAR L) (CAR (CDR L))))))

Terminology. The *standard alist* for a sequence of variable symbols **args** is **NIL** if **args** is empty and otherwise is (CONS (CONS 'v v) **alist**) where **v** is the first symbol in **args**, 'v is the **LITATOM** corresponding to **v**, and **alist** is the standard alist for the sequence of symbols obtained by deleting **v** from **args**.

Example. The standard alist for **X**, **ANS**, and **L** is

```
(LIST (CONS 'X X)
      (CONS 'ANS ANS)
      (CONS 'L L))
```

Terminology. The *non-SUBRP axiom* for **fn**, **args**, and **body**, where **fn** is a function symbol, **args** is a sequence of variable symbols, and **body** is a term, is

```
(AND (EQUAL (SUBRP 'fn) F)
      (EQUAL (FORMALS 'fn) eargs)
      (EQUAL (BODY 'fn) ebody))
```

where **'fn** is the **LITATOM** corresponding to **fn**, **eargs** is the quotation list for **args**, and **ebody** is the preferred quotation of **body** unless **body** has the form **(EVAL\$ flg ebody1 alist)** where

1. **flg** is an explicit value other than **'LIST**;
2. **ebody1** is an explicit value that is a quotation of some term **body1**;
3. **alist** is the standard alist for **args**; and
4. the set of variables in **body1** is a subset of those in **args**,

in which case **ebody** is the preferred quotation of **body1**.

Examples. The non-SUBRP axiom for **ADD2**, **(X Y)**, and **(PLUS 2 X Y)** is

```
(AND (EQUAL (SUBRP 'ADD2) F)
      (EQUAL (FORMALS 'ADD2) '(X Y))
      (EQUAL (BODY 'ADD2)
              '(PLUS (QUOTE 2) (PLUS X Y)))).
```

The non-SUBRP axiom for **RUS**, **()**, and

```
(EVAL$ T '(ADD1 (RUS)) NIL)
```

is

```
(AND (EQUAL (SUBRP 'RUS) F)
      (EQUAL (FORMALS 'RUS) NIL)
      (EQUAL (BODY 'RUS) '(ADD1 (RUS)))).
```

4.11. Quantification

Note. The reader is urged to see [30] for a motivated development of our axiomatization of **FOR**.

4.11.1. The Definition of *FOR* and its Subfunctions

Defining Axiom 56.

```
(QUANTIFIER-INITIAL-VALUE OP)
=
(CDR (ASSOC OP '( (ADD-TO-SET . NIL)
                  (ALWAYS . *1*TRUE)
                  (APPEND . NIL)
                  (COLLECT . NIL)
                  (COUNT . 0)
                  (DO-RETURN . NIL)
                  (EXISTS . *1*FALSE)
                  (MAX . 0)
                  (SUM . 0)
                  (MULTIPLY . 1)
                  (UNION . NIL))))))
```

Defining Axiom 57.

```
(QUANTIFIER-OPERATION OP VAL REST)
=
(IF (EQUAL OP 'ADD-TO-SET) (ADD-TO-SET VAL REST)
  (IF (EQUAL OP 'ALWAYS)   (AND VAL REST)
    (IF (EQUAL OP 'APPEND)  (APPEND VAL REST)
      (IF (EQUAL OP 'COLLECT) (CONS VAL REST)
        (IF (EQUAL OP 'COUNT) (IF VAL (ADD1 REST) REST)
          (IF (EQUAL OP 'DO-RETURN) VAL
            (IF (EQUAL OP 'EXISTS) (OR VAL REST)
              (IF (EQUAL OP 'MAX) (MAX VAL REST)
                (IF (EQUAL OP 'SUM) (PLUS VAL REST)
                  (IF (EQUAL OP 'MULTIPLY) (TIMES VAL REST)
                    (IF (EQUAL OP 'UNION) (UNION VAL REST)
                      0)))))))))))))
```

Defining Axiom 58.

```
(FOR V L COND OP BODY A)
=
(IF (NLISTP L)
  (QUANTIFIER-INITIAL-VALUE OP)
  (IF (EVAL$ T COND (CONS (CONS V (CAR L)) A))
    (QUANTIFIER-OPERATION OP
      (EVAL$ T BODY (CONS (CONS V (CAR L)) A))
      (FOR V (CDR L) COND OP BODY A))
    (FOR V (CDR L) COND OP BODY A)))
```

4.11.2. The Extended Syntax for FOR—Abbreviations II

Note. This section completes the precise specification of the extended syntax by defining when an s-expression is an “abbreviated **FOR**” and the “**FOR** expression denoted” by such an s-expression.

Terminology. An s-expression **x** of the form (**FOR v IN lst WHEN cond op body**)—i.e., **x** is an s-expression of length eight whose first element is the word **FOR**, third element is the word **IN**, and fifth element is the word **WHEN**—is an *abbreviated FOR* iff each of the following is true:

- **v** is a variable symbol,
- **lst**, **cond**, and **body** are well-formed s-expressions whose translations are the terms **t-lst**, **t-cond**, and **t-body**, and
- **op** is an element of the set {**ADD-TO-SET ALWAYS APPEND COLLECT COUNT DO-RETURN EXISTS MAX SUM MULTIPLY UNION**}.

The *FOR expression denoted* by such an **x** is (**FOR 'v t-lst 't-cond 'op 't-body alist**) where **'v**, **'t-cond**, **'op**, and **'t-body** are the preferred quotations (see page 170) of **v**, **t-cond**, **op**, and **t-body** respectively, and **alist** is the standard alist (see page 175) on the sequence of variable symbols obtained by deleting **v** from the union of the variable symbols of **t-cond** with those of **t-body** and then sorting the resulting set alphabetically. An s-expression of the form (**FOR x IN lst op body**) is an *abbreviated FOR* iff (**FOR x IN lst WHEN T op body**) is an abbreviated **FOR** and, if so, denotes the same **FOR** expression as that denoted by (**FOR x IN lst WHEN T op body**). No other form of s-expression is an *abbreviated FOR*.

4.12. SUBRPs and non-SUBRPs

Note. The symbol **QUOTE**, which is treated specially by **V&C\$** and cannot be defined by the user, is not a **SUBRP**.

Axiom 59.

(NOT (SUBRP 'QUOTE)).

Axioms 60-64. We now add the non-**SUBRP** axiom for each of the following five function symbols: **APPLY\$**, **EVAL\$**, **V&C\$**, **V&C-APPLY\$**, and **FOR**. Each of these symbols was introduced with a defining axiom of the form **(fn $x_1 \dots x_n$) = body**. For each of the five function symbols we add the non-**SUBRP** axiom for **fn**, **($x_1 \dots x_n$)**, and **body**.

Axioms 65-121. We add the **SUBRP** axiom for every other function symbol that is mentioned in an axiom of the current theory. The complete list of **SUBRPs** is: **ADD1**, **ADD-TO-SET**, **AND**, **APPEND**, **APPLY-SUBR**, **ASSOC**, **BODY**, **CAR**, **CDR**, **CONS**, **COUNT**, **DIFFERENCE**, **EQUAL**, **FALSE**, **FALSEP**, **FIX**, **FIX-COST**, **FORMALS**, **GEQ**, **GREATERP**, **IDENTITY**, **IF**, **IFF**, **IMPLIES**, **LEQ**, **LESSP**, **LISTP**, **LITATOM**, **MAX**, **MEMBER**, **MINUS**, **NEGATIVEP**, **NEGATIVE-GUTS**, **NLISTP**, **NOT**, **NUMBERP**, **OR**, **ORDINALP**, **ORD-LESSP**, **PACK**, **PAIRLIST**, **PLUS**, **QUANTIFIER-INITIAL-VALUE**, **QUANTIFIER-OPERATION**, **QUOTIENT**, **REMAINDER**, **STRIP-CARS**, **SUB1**, **SUBRP**, **SUM-CDRS**, **TIMES**, **TRUE**, **TRUEP**, **UNION**, **UNPACK**, **ZERO**, and **ZEROP**.

4.13. Induction and Recursion

4.13.1. Induction

Rule of Inference. Induction

Suppose

- (a) **p** is a term;
- (b) **m** is a term;
- (c) **q₁**, ..., **q_k** are terms;
- (d) **h₁**, ..., **h_k** are positive integers;
- (e) it is a theorem that **(ORDINALP m)**;
- and
- (f) for $1 \leq i \leq k$ and $1 \leq j \leq h_i$, **s_{i,j}** is a substitution and it is a theorem that

(IMPLIES q_i (ORD-LESSP m/s_{i,j} m)).

Then \mathbf{p} is a theorem if

$$(1) \text{ (IMPLIES (AND (NOT } \mathbf{q}_1) \dots (\text{NOT } \mathbf{q}_k)) \otimes \mathbf{T} \\ \mathbf{p})}$$

is a theorem and

for each $1 \leq i \leq k$,

$$(2) \text{ (IMPLIES (AND } \mathbf{q}_i \text{ } \mathbf{p}/\mathbf{s}_{i,1} \dots \mathbf{p}/\mathbf{s}_{i,h_i}) \otimes \mathbf{T} \\ \mathbf{p})}$$

is a theorem.

Notes. In informally describing an application of the induction principle to some conjecture \mathbf{p} we generally say the induction is *on* the variables occurring in the term \mathbf{m} , which is called the *measure*. An inductive proof splits the problem into $k+1$ cases, a *base case*, given by formula (1) above, and k *induction steps*, given by the k formulas of form (2) above. The *cases* are given by the \mathbf{q}_i . The i^{th} induction step requires proving \mathbf{p} under case \mathbf{q}_i . The base case requires proving \mathbf{p} under the conjunction of the negations of the \mathbf{q}_i . In the i^{th} induction step one may assume an arbitrary number (namely h_i) of instances, $\mathbf{p}/\mathbf{s}_{i,j}$, of the conjecture being proved. The j^{th} instance for the i^{th} case is given by substitution $\mathbf{s}_{i,j}$. Each instance is called an *induction hypothesis*. To *justify* an induction one must show in the theorems of supposition (f) that some ordinal measure \mathbf{m} of the induction variables decreases under each substitution in each respective case.

We illustrate the principle of induction in Chapter 5.

4.13.2. The Principle of Definition

Terminology. We say that a term \mathbf{t} *governs* an occurrence of a term \mathbf{s} in a term \mathbf{b} iff (a) either \mathbf{b} contains a subterm of the form $(\mathbf{IF} \ \mathbf{t} \ \mathbf{p} \ \mathbf{q})$ and the occurrence of \mathbf{s} is in \mathbf{p} or (b) if \mathbf{b} contains a subterm of the form $(\mathbf{IF} \ \mathbf{t}' \ \mathbf{p} \ \mathbf{q})$, where \mathbf{t} is $(\mathbf{NOT} \ \mathbf{t}')$ and the occurrence of \mathbf{s} is in \mathbf{q} .

Examples. The terms **P** and (**NOT Q**) govern the first occurrence of **S** in

```
( IF P
  ( IF ( IF Q A S )
        S
      B )
    C )
```

The terms **P** and (**IF Q A S**) govern the second occurrence of **S**.

Note. The mechanization of the logic is slightly more restrictive because it only inspects the “top-level” **IF**s in **b**. Thus, the mechanization recognizes that **P** governs **S** in (**IF P (FN (IF Q S A)) B**) but it does not recognize that **Q** governs **S** also.

Extension Principle. Definition

The axiomatic act

Definition. (**f x₁ ... x_n**) = **body**

is admissible under the history **h** provided

- (a) **f** is a function symbol of **n** arguments and
is new in **h**;
- (b) **x₁, ..., x_n** are distinct variables;
- (c) **body** is a term and mentions no symbol as a
variable other than **x₁, ..., x_n**;
- (d) there is a term **m** such that (a) (**ORDINALP m**)
can be proved directly in **h**, and (b) for each
occurrence of a subterm of the form (**f y₁ ... y_n**)
in **body** and the terms **t₁, ..., t_k** governing it,
the following formula can be proved directly in **h**:

$$(\text{IMPLIES } (\text{AND } t_1 \dots t_k) \otimes T \\ (\text{ORD-LESSP } m/s \ m)) ,$$

where **s** is the substitution {<**x₁**, **y₁**> ... <**x_n**, **y_n**>}.

If admissible, we add the

Defining Axiom.

$(f\ x_1 \dots x_n) = \text{body}.$

In addition, we add

Axiom.

the non-SUBRP axiom for f , (x_1, \dots, x_n) , and body .

4.14. Constraints and Functional Instantiation

Notes. In this section we describe another extension principle and a new rule of inference: the introduction of function symbols by “constraint” and derivation by “functional instantiation.” The validity of these principles can be derived from the foregoing material, but because they change the “flavor” of the logic by permitting certain apparently higher-order acts we prefer to include their description here.

Functional instantiation permits one to infer new theorems from old ones by instantiating function symbols instead of variables. To be sure that such an instantiation actually produces a theorem, we first check that the formulas that result from similarly instantiating certain of the axioms about the function symbols being replaced are also theorems. Intuitively speaking, the correctness of this derived rule of inference consists of little more than the trivial observation that one may systematically change the name of a function symbol to a new name in a first-order theory without losing any theorems, modulo the renaming. However, we have found that this trivial observation has important potential practical ramifications in reducing mechanical proof efforts. We also find that this observation leads to superficially shocking results, such as the proof of the associativity of **APPEND** by instantiation rather than induction. And finally, we are intrigued by the extent to which such techniques permit one to capture the power of higher-order logic within first-order logic.

In order to make effective use of functional instantiation, we have found it necessary to augment the logic with an extension principle that permits the introduction of new function symbols without completely characterizing them. This facility permits one to add axioms about a set of new function symbols, provided one can exhibit “witnessing” functions that have the alleged properties. The provision for witnesses ensures that the new axioms do not render the logic inconsistent.

An example of the use of such constraints is to introduce a two-place function symbol, **FN**, constrained to be commutative. Any commutative function, e.g., a

constant function of two arguments, suffices to witness the new axiom about **FN**. One can then prove theorems about **FN**. These theorems necessarily depend upon only one fact about **FN**, the fact that it is commutative. Thus, no matter how complicated these theorems are to prove, the analogous theorems about some other function symbol can be inferred via functional instantiation at the mere cost of confirming that the new symbol is commutative.

In the paper [19] we derive introduction by constraint and functional instantiation and we show many examples of their use, including formal studies of “mapping functions” such as **FOLDR** and **FOLDL**, alternatives to quantifiers, and properties of generic language interpreters. These examples are also included among the example files of Nqthm-1992. See example file 7.

Terminology. A *LAMBDA expression* is an s-expression of the form **(LAMBDA (a₁ ... a_n) body)**, where the **a_i** are distinct variable symbols and **body** is a term. The *arity* of such a **LAMBDA** expression is **n**, its *argument list* is the sequence **a₁, ..., a_n**, and its *body* is **body**.

Terminology. A *functional substitution* is a function on a finite set of function symbols such that for each pair **<f, g>** in the substitution, **g** is either a function symbol or **LAMBDA** expression and the arity of **f** is the arity of **g**.

Terminology. We recursively define the *functional instantiation of a term t under a functional substitution fs*. If **t** is a variable, the result is **t**. If **t** is the term **(f t₁ ... t_n)**, let **t_i'** be the functional instantiation of **t_i**, for **i** from **1** to **n** inclusive, under **fs**. If, for some function symbol **f'**, the pair **<f, f'>** is in **fs**, the result is **(f' t₁' ... t_n')**. If a pair **<f, (LAMBDA (a₁ ... a_n) term)>** is in **fs**, the result is **term/{... <a_i, t_i'> ...}**. Otherwise, the result is **(f t₁' ... t_n')**.

Note. Recall that “**term/σ**” denotes the result of applying the ordinary (variable) substitution **σ** to **term**. If **σ** is the variable substitution **{<X, (FN A)> <Y, B>}**, then **(PLUS X Y)/σ** is **(PLUS (FN A) B)**.

Example. The functional instantiation of the term **(PLUS (FN X) (TIMES Y Z))** under the functional substitution **{<PLUS, DIFFERENCE> <FN, (LAMBDA (V) (QUOTIENT V A))>}** is the term **(DIFFERENCE (QUOTIENT X A) (TIMES Y Z))**.

Terminology. We recursively define the *functional instantiation of a formula φ under a functional substitution fs*. If **φ** is **φ₁ ∨ φ₂**, then the result is **φ₁' ∨ φ₂'**, where **φ₁'** and **φ₂'** are the functional instantiations of **φ₁** and **φ₂** under **fs**. If **φ** is **¬φ₁**, then the result is **¬φ₁'**, where **φ₁'** is the functional instantiation of **φ₁**

under fs. If ϕ is $(\mathbf{x} = \mathbf{y})$, then the result is $(\mathbf{x}' = \mathbf{y}')$, where \mathbf{x}' and \mathbf{y}' are the functional instantiations of \mathbf{x} and \mathbf{y} under fs.

Terminology. A variable \mathbf{v} is said to be *free* in $(\mathbf{LAMBDA} (\mathbf{a}_1 \dots \mathbf{a}_n) \mathbf{term})$ if and only if \mathbf{v} is a variable of \mathbf{term} and \mathbf{v} is not among the \mathbf{a}_i . A variable \mathbf{v} is said to be *free* in a functional substitution if and only if it is free in a **LAMBDA** expression in the range of the substitution. A variable \mathbf{v} is said to be *bound* in $(\mathbf{LAMBDA} (\mathbf{a}_1 \dots \mathbf{a}_n) \mathbf{term})$ if and only if \mathbf{v} is among the \mathbf{a}_i .

Terminology. We denote functional instantiation with \backslash to distinguish it from ordinary (variable) substitution, which is denoted with $/$.

Example. If ρ is the functional substitution $\{<\mathbf{PLUS}, (\mathbf{LAMBDA} (\mathbf{U} \mathbf{V}) (\mathbf{ADD1} \mathbf{U}))>\}$ then $(\mathbf{PLUS} \mathbf{X} \mathbf{Y})\backslash\rho$ is $(\mathbf{ADD1} \mathbf{X})$.

Extension Principle. Conservatively constraining new function symbols.

The axiomatic act

Constraint.

Constrain \mathbf{f}_1, \dots , and \mathbf{f}_n to have property \mathbf{ax} ,

is admissible under the history h provided there exists a functional substitution fs with domain $\{\mathbf{f}_1 \dots \mathbf{f}_n\}$ such that

1. the \mathbf{f}_i are distinct new function symbols,
2. each member of the range of fs is either an old function symbol or is a **LAMBDA** expression whose body is formed of variables and old function symbols,
3. no variable is free in fs, and
4. $\mathbf{ax}\backslash\text{fs}$ is a theorem of h .

If admissible, the act adds the axiom \mathbf{ax} to the history h . The image of \mathbf{f}_i under fs is called the *witness* for \mathbf{f}_i .

Note. Unlike the shell and definitional events, the constraint event does not add any **SUBRP** or non-**SUBRP** axioms about the new function symbols.

Terminology. A functional substitution fs is *tolerable* with respect to a history h provided that the domain of fs contains only function symbols introduced into h by the user via extension principles other than the shell mechanism.

Notes. We do not want to consider functionally substituting for built-in function symbols or shell function symbols because the axioms about them are so diffuse in the implementation. We especially do not want to consider substituting for such function symbols as **ORD-LESSP**, because they are used in the principle of induction.

The rule of functional instantiation requires that we prove appropriate functional instances of certain axioms. Roughly speaking, we have to prove that the “new” functions satisfy all the axioms about the “old” ones. However we must be careful to avoid “capture”-like problems. For example, if the axiom constraining an old function symbol happens to involve a variable that is free in the functional substitution, then the functional instantiation of that axiom may be a weaker formula than the soundness argument in [19] requires. We illustrate this problem in the example below.

Example. Imagine that **ID1** is a function of two arguments that always returns its first argument. Let the defining axiom for **ID1** be the term **(EQUAL (ID1 X Y) X)**. Call this term **t**. Let **fs** be the functional substitution $\{ \langle \text{ID1}, (\text{LAMBDA (A B) X}) \rangle \}$. This substitution replaces **ID1** by the constant function that returns **X**. Observe that the **fs** instance of the defining axiom for **ID1** is a theorem, i.e., **t****fs** is **(EQUAL X X)**. A careless definition of the rule of functional instantiation would therefore permit the conclusion that any fact about **ID1** holds for the constant function that returns **X**. But this conclusion is specious: **(EQUAL (ID1 (ADD1 X) Y) (ADD1 X))** is a fact about **ID1** but its functional instance under **fs**, **(EQUAL X (ADD1 X))**, is not. What is wrong? A variable in the defining axiom, **t**, occurs freely in **fs**. We were just “lucky” that the **fs** instance of the defining axiom, **t**, was a theorem. Had **ID1** been defined with the equivalent axiom **(EQUAL (ID1 U V) U)**, which we will call **t'**, we would have been required to prove **t'****fs** which is **(EQUAL X U)** and is unprovable. The point is that when we attempt to prove the functional instances of the axioms, we must first rename the variables in the axioms so as to avoid the free variables in the functional substitution.

Terminology. Term **t**₁ is a *variant* of term **t**₂ if **t**₁ is an instance of **t**₂ and **t**₂ is an instance of **t**₁.

Example. **(EQUAL (ID1 U V) U)** is a variant of **(EQUAL (ID1 X Y) X)**.

Terminology. If **fs** is a functional substitution and **t** is a term, then *an fs renaming of t* is any variant of **t** containing no variable free in **fs**.

Derived Rule Of Inference. Functional Instantiation.

If h is a history, fs is a tolerable functional substitution, \mathbf{thm} is a theorem of h , and the fs instance of an fs renaming of every axiom of h can be proved in h , then \mathbf{thm}/fs is a theorem of a definitional/constrain extension of h .

Note. In [19] we show that the introduction of constrained function symbols preserves the consistency of the logic and we derive the rule of functional instantiation. In fact, we prove a more general version of it that permits us to ignore the “irrelevant” axioms and definitions in h .

5 Proving Theorems in the Logic

We have already proved many theorems in the logic. For example, from

Defining Axiom 36.

```
(APPEND L1 L2)
=
(IF (LISTP L1)
    (CONS (CAR L1) (APPEND (CDR L1) L2))
    L2)
```

we have observed that

$$(\text{APPEND NIL } '(1\ 2\ 3)) = '(1\ 2\ 3)$$

and

$$(\text{APPEND } '(1\ 2\ 3) \ '(4\ 5\ 6)) = '(1\ 2\ 3\ 4\ 5\ 6)$$

These two equations are *theorems* that can be *proved* using the familiar rules of inference of propositional calculus, equality, and instantiation.

In this Chapter we illustrate many simple proofs. We start with the propositional calculus and work our way up to inductive proofs about recursive functions. Among the illustrative theorems proved are those that justify “structural induction,” the most commonly used application of the induction principle in our logic.

5.1. Propositional Calculus with Equality

We will start with some simple propositional calculus. Most of the proofs in this section were contributed by N. Shankar. First, we will prove the theorem $(\mathbf{P} \vee \mathbf{Q}) \rightarrow (\mathbf{Q} \vee \mathbf{P})$. However, the formal proof of even so simple a theorem is very long if we limit ourselves to the rules of inference given, namely Expansion, Contraction, Associativity, and Cut. We therefore start by deriving several new rules of inference.

Derived Rule of Inference. *Commutativity of Or:*

Derive $(\mathbf{b} \vee \mathbf{a})$ from $(\mathbf{a} \vee \mathbf{b})$.

Below we derive $(\mathbf{b} \vee \mathbf{a})$ from $(\mathbf{a} \vee \mathbf{b})$. The proof is presented as a sequence of formulas, each of which is either an axiom or is given (in the case of a derived rule of inference) or is derived by inference rules from previous formulas. We number and justify each formula in the presentation below. Unless otherwise noted, each rule of inference takes as its single premise the formula immediately above.

Proof.

- | | | |
|----|--------------------------------------|---------------------|
| 1. | $(\mathbf{a} \vee \mathbf{b})$ | Given |
| 2. | $(\neg(\mathbf{a}) \vee \mathbf{a})$ | Propositional Axiom |
| 3. | $(\mathbf{b} \vee \mathbf{a})$ | Cut, lines 1 and 2 |
| | Q.E.D. | |

What exactly is going on here? We have shown that from $(\mathbf{a} \vee \mathbf{b})$ we can derive $(\mathbf{b} \vee \mathbf{a})$ by applying the primitive rules, namely the Propositional Axiom and Cut. Thus, in the future, if we have derived a formula of the form $(\mathbf{a} \vee \mathbf{b})$ we can, in the next line, derive $(\mathbf{b} \vee \mathbf{a})$ and attribute the derivation to the derived rule Commutativity of Or, in the knowledge that we could convert the resulting “proof” into a proof by substituting for that single line the steps above.

We now derive several other useful rules, using Commutativity of Or:

Derived Rule of Inference. *Or Insertion 1:*

Derive $\mathbf{a} \vee (\mathbf{c} \vee \mathbf{b})$ from $(\mathbf{a} \vee \mathbf{b})$.

Proof.

- | | | |
|----|--|---------------------|
| 1. | $(\mathbf{a} \vee \mathbf{b})$ | Given |
| 2. | $(\mathbf{b} \vee \mathbf{a})$ | Commutativity of Or |
| 3. | $(\mathbf{c} \vee (\mathbf{b} \vee \mathbf{a}))$ | Expansion |
| 4. | $(\mathbf{c} \vee \mathbf{b}) \vee \mathbf{a}$ | Associativity |
| 5. | $\mathbf{a} \vee (\mathbf{c} \vee \mathbf{b})$ | Commutativity of Or |
| | Q.E.D. | |

Derived Rule of Inference. *Or Insertion 2:*

Derive $\mathbf{a} \vee (\mathbf{b} \vee \mathbf{c})$ from $(\mathbf{a} \vee \mathbf{b})$.

Proof.

- | | | |
|----|---------------------|---------------------|
| 1. | $(a \vee b)$ | Given |
| 2. | $(b \vee a)$ | Commutativity of Or |
| 3. | $b \vee (c \vee a)$ | Or Insertion 1 |
| 4. | $(b \vee c) \vee a$ | Associativity |
| 5. | $a \vee (b \vee c)$ | Commutativity of Or |
- Q.E.D.

Derived Rule of Inference. Or-Implication:

Derive $(a \vee b) \rightarrow c$ from $a \rightarrow c$ and $b \rightarrow c$.

Proof.

- | | | |
|-----|--|---------------------|
| 1. | $b \rightarrow c$ | Given |
| 2. | $\neg b \vee c$ | Abbreviation |
| 3. | $\neg(a \vee b) \vee (a \vee b)$ | Propositional Axiom |
| 4. | $(\neg(a \vee b) \vee a) \vee b$ | Associativity |
| 5. | $b \vee (\neg(a \vee b) \vee a)$ | Commutativity of Or |
| 6. | $(\neg(a \vee b) \vee a) \vee c$ | Cut, lines 5 and 2 |
| 7. | $c \vee (\neg(a \vee b) \vee a)$ | Commutativity of Or |
| 8. | $(c \vee \neg(a \vee b)) \vee a$ | Associativity |
| 9. | $a \vee (c \vee \neg(a \vee b))$ | Commutativity of Or |
| 10. | $a \rightarrow c$ | Given |
| 11. | $\neg a \vee c$ | Abbreviation |
| 12. | $\neg a \vee (c \vee \neg(a \vee b))$ | Or Insertion 2 |
| 13. | $(c \vee \neg(a \vee b)) \vee (c \vee \neg(a \vee b))$ | Cut, lines 9 and 12 |
| 14. | $(c \vee \neg(a \vee b))$ | Contraction |
| 15. | $\neg(a \vee b) \vee c$ | Commutativity of Or |
| 16. | $(a \vee b) \rightarrow c$ | Abbreviation |
- Q.E.D.

We are finally in a position to prove our first theorem!

Theorem. $(P \vee Q) \rightarrow (Q \vee P)$.**Proof.**

- | | | |
|----|-------------------------------------|-------------------------------|
| 1. | $\neg P \vee P$ | Propositional Axiom |
| 2. | $\neg P \vee (Q \vee P)$ | Or Insertion 1 |
| 3. | $P \rightarrow (Q \vee P)$ | Abbreviation |
| 4. | $\neg Q \vee Q$ | Propositional Axiom |
| 5. | $\neg Q \vee (Q \vee P)$ | Or Insertion 2 |
| 6. | $Q \rightarrow (Q \vee P)$ | Abbreviation |
| 7. | $(P \vee Q) \rightarrow (Q \vee P)$ | Or-Implication, lines 3 and 6 |
- Q.E.D.

A very useful derived rule is, of course,

Derived Rule of Inference. *Modus Ponens*:

Derive **b** from **a** and **(a → b)**.

Proof.

- | | | |
|----|---------------|--------------------|
| 1. | a | Given |
| 2. | a ∨ b | Expansion |
| 3. | a → b | Given |
| 4. | ¬a ∨ b | Abbreviation |
| 5. | b ∨ b | Cut, lines 2 and 4 |
| 6. | b | Contraction |
- Q.E.D.

There are, of course, many other derived rules of inference about propositional calculus. Among them are versions of the tautology theorem, the deduction law, and case analysis. We will use such rules freely below and urge readers unfamiliar with them to consult a logic textbook, e.g., [130].

We next prove a simple theorem about the equality predicate. We are not so much interested in the theorem as in the proof, because it suggests the proof of the commonly used derived rule of substitution of equals for equals.

Theorem.

X=Y → (CAR (CDR X))=(CAR (CDR Y))

Proof.

- | | | |
|----|---|------------------------|
| 1. | X1=Y1 → (CDR X1)=(CDR Y1) | Equality Axiom for CDR |
| 2. | X=Y → (CDR X)=(CDR Y) | Instantiation |
| 3. | ¬(X=Y) ∨ (CDR X)=(CDR Y) | Abbreviation |
| 4. | (CDR X)=(CDR Y) ∨ ¬(X=Y) | Commutativity of Or |
| 5. | X1=Y1 → (CAR X1)=(CAR Y1) | Equality Axiom for CAR |
| 6. | (CDR X)=(CDR Y) → (CAR (CDR X))=(CAR (CDR Y)) | Instantiation |
| 7. | ¬((CDR X)=(CDR Y)) ∨ (CAR (CDR X))=(CAR (CDR Y)) | Abbreviation |
| 8. | ¬(X=Y) ∨ (CAR (CDR X))=(CAR (CDR Y)) | Cut, lines 4 and 7 |
| 9. | X=Y → (CAR (CDR X))=(CAR (CDR Y)) | Abbreviation |
- Q.E.D.

By induction on the structure of terms we can derive the following powerful and widely used rule of inference:

Derived Rule of Inference. *Substitution of Equals for Equals:*

If $\mathbf{t}=\mathbf{s}$ has been proved and formula \mathbf{q} is obtained from formula \mathbf{p} by replacing some occurrences of \mathbf{t} by \mathbf{s} , then \mathbf{q} is provable iff \mathbf{p} is provable.

For example, this rule lets us replace the application of a function, e.g., $(\mathbf{FIX} \mathbf{A})$, by the instantiated body of the function, $(\mathbf{IF} (\mathbf{NUMBERP} \mathbf{A}) \mathbf{A} \mathbf{0})$, anywhere $(\mathbf{FIX} \mathbf{A})$ occurs in another formula.

By combining this rule with the deduction law we can substitute \mathbf{s} for selected occurrences of \mathbf{t} in **concl** when trying to prove a formula of the form $\mathbf{t} = \mathbf{s} \rightarrow \mathbf{concl}$.

5.2. Theorems about IF and Propositional Functions

Theorem. **IF cases:**

$$(\mathbf{IF} \mathbf{X} \mathbf{Y} \mathbf{Z})=\mathbf{U} \quad \leftrightarrow \quad (\mathbf{X} \neq \mathbf{F} \rightarrow \mathbf{Y}=\mathbf{U}) \wedge (\mathbf{X}=\mathbf{F} \rightarrow \mathbf{Z}=\mathbf{U})$$

Proof. The proof is by case analysis on $\mathbf{X}=\mathbf{F}$.

Case 1. $\mathbf{X} = \mathbf{F}$. By **Axiom 4**, $(\mathbf{IF} \mathbf{X} \mathbf{Y} \mathbf{Z}) = \mathbf{Z}$. Thus, the left-hand side above is $\mathbf{Z}=\mathbf{U}$. The right-hand side becomes

$$(\mathbf{F} \neq \mathbf{F} \rightarrow \mathbf{Y}=\mathbf{U}) \wedge (\mathbf{F}=\mathbf{F} \rightarrow \mathbf{Z}=\mathbf{U})$$

By propositional calculus and equality, the right-hand side is equivalent to $\mathbf{Z}=\mathbf{U}$.

Case 2. $\mathbf{X} \neq \mathbf{F}$. By **Axiom 5**, the left-hand side above is equivalent to $\mathbf{Y}=\mathbf{U}$. By propositional calculus and equality, the right-hand side also equivalent to $\mathbf{Y}=\mathbf{U}$. Q.E.D.

Theorem. **IMPLIES is implication:**

$$(\mathbf{IMPLIES} \mathbf{P} \mathbf{Q}) \neq \mathbf{F} \quad \leftrightarrow \quad (\mathbf{P} \neq \mathbf{F} \rightarrow \mathbf{Q} \neq \mathbf{F})$$

Proof. We will repeatedly replace the left-hand side of this equivalence by equivalent formulas until it is identical to the right-hand side.

By the definition of **IMPLIES** (**Axiom 11**), the left-hand side is equivalent to

$$(\mathbf{IF} \mathbf{P} (\mathbf{IF} \mathbf{Q} \mathbf{T} \mathbf{F}) \mathbf{T}) \neq \mathbf{F}$$

which, by **IF cases** is equivalent to

$$(\mathbf{P} \neq \mathbf{F} \rightarrow (\mathbf{IF} \mathbf{Q} \mathbf{T} \mathbf{F}) \neq \mathbf{F}) \wedge (\mathbf{P}=\mathbf{F} \rightarrow \mathbf{T} \neq \mathbf{F}).$$

By **Axiom 1** and propositional calculus, the above is equivalent to

$$\mathbf{P} \neq \mathbf{F} \rightarrow (\mathbf{IF} \mathbf{Q} \mathbf{T} \mathbf{F}) \neq \mathbf{F}.$$

Applying **IF cases** again produces

$$P \neq F \rightarrow [(Q \neq F \rightarrow T \neq F) \wedge (Q = F \rightarrow F \neq F)]$$

which, by **Axiom 1** and propositional calculus with equality, is equivalent to

$$(P \neq F \rightarrow Q \neq F).$$

Q.E.D.

Next we observe

Derived Rule of Inference. *Term based Modus Ponens:*

The term **q** may be derived as a theorem if the term **p** is a theorem and the term **(IMPLIES p q)** is a theorem.

The proof is trivial, given **IMPLIES is implies**.

Theorem. NOT case:

$$(\text{NOT } P) \neq F \leftrightarrow P = F$$

Proof. By the definition of **NOT (Axiom 8)**, the left-hand side is equivalent to **(IF P F T) ≠ F** which, by **IF cases**, is equivalent to

$$(P \neq F \rightarrow F \neq F) \wedge (P = F \rightarrow T \neq F).$$

But by **Axiom 1** and propositional calculus with equality, this is equivalent to **P = F**. Q.E.D.

Theorem. EQUAL is equal:

$$(\text{EQUAL } X \ Y) \neq F \leftrightarrow X = Y$$

Proof. That the left-hand side implies the right is merely the contrapositive of

Axiom 3.

$$X \neq Y \rightarrow (\text{EQUAL } X \ Y) = F.$$

That the right-hand side implies the left follows immediately from

Axiom 2.

$$X = Y \rightarrow (\text{EQUAL } X \ Y) = T$$

and

Axiom 1.

$$T \neq F.$$

by equality reasoning. Q.E.D.

Henceforth, we will reason about the propositional functions **EQUAL**, **IF**, **NOT**, **AND**, **OR**, and **IMPLIES** with the same casual ease we use with propositional calculus and equality.

5.3. Simple Theorems about NIL, CONS, and APPEND

Theorem. **NIL is not a LISTP:**
(LISTP NIL)=F

Informally, the proof of this theorem is “**NIL** is an abbreviation for a **PACK** expression, **PACK** returns a **LITATOM**, and no **LITATOM** is a **LISTP**.” We do it more formally below.

Proof. Recall that **NIL** is an abbreviation for

(PACK '(78 73 76 . 0))

Thus, by instantiating

Axiom 20.2.
(LITATOM (PACK X1)),

replacing **X1** by **'(78 73 76 . 0)**, we get

(1) (LITATOM NIL)

By instantiating

Axiom 20.8.1.
(IMPLIES (LITATOM X) (NOT (LISTP X)))

replacing **X** by **NIL**, we get

(2) (IMPLIES (LITATOM NIL) (NOT (LISTP NIL)))

Term based modus ponens on lines (1) and (2) produces

(3) (NOT (LISTP NIL)).

By **NOT case**, instantiating **P** with **(LISTP NIL)** (and choosing only the “ \rightarrow ” direction of the “ \leftrightarrow ”) we get

(4) (NOT (LISTP NIL)) \rightarrow (LISTP NIL)=F.

Thus, with modus ponens on lines (3) and (4) we get

(LISTP NIL) = F. Q.E.D.

We next prove two lemmas that make it easy to “compute” **APPEND** expressions:

Theorem. APPEND-NIL:
 $(\text{APPEND NIL L}) = L$

Proof. Instantiating the definition of **APPEND**,

Defining Axiom 36.

$$\begin{aligned} &(\text{APPEND L1 L2}) \\ &= \\ &(\text{IF (LISTP L1)} \\ &\quad (\text{CONS (CAR L1) (APPEND (CDR L1) L2)}) \\ &\quad \text{L2}) \end{aligned}$$

replacing **L1** by **NIL** and **L2** by **L** we get

$$\begin{aligned} &(\text{APPEND NIL L}) \\ &= \\ &(\text{IF (LISTP NIL)} \\ &\quad (\text{CONS (CAR NIL) (APPEND (CDR NIL) L)}) \\ &\quad \text{L}). \end{aligned}$$

By **NIL is not a LISTP** and substitution of equals for equals we get

$$\begin{aligned} &(\text{APPEND NIL L}) \\ &= \\ &(\text{IF F} \\ &\quad (\text{CONS (CAR NIL) (APPEND (CDR NIL) L)}) \\ &\quad \text{L}). \end{aligned}$$

By instantiation of

Axiom 4.

$$X = F \rightarrow (\text{IF X Y Z}) = Z$$

and equality we get

$$\begin{aligned} &(\text{IF F} \\ &\quad (\text{CONS (CAR NIL) (APPEND (CDR NIL) L)}) \\ &\quad \text{L}) \\ &= \\ &\text{L.} \end{aligned}$$

Thus, by equality,

$$(\text{APPEND NIL L}) = L.$$

Q.E.D.

The second lemma about **APPEND** is

Theorem. **APPEND-CONS:**
 $(\text{APPEND } (\text{CONS } X \ Y) \ L) = (\text{CONS } X \ (\text{APPEND } Y \ L)).$

Proof. By instantiation of **Defining Axiom 36** (above)

$$\begin{aligned} & (\text{APPEND } (\text{CONS } X \ Y) \ L) \\ & \quad = \\ & (\text{IF } (\text{LISTP } (\text{CONS } X \ Y)) \\ & \quad \quad (\text{CONS } (\text{CAR } (\text{CONS } X \ Y)) \\ & \quad \quad \quad (\text{APPEND } (\text{CDR } (\text{CONS } X \ Y)) \ L)) \\ & \quad \quad L). \end{aligned}$$

But, by

Axiom 19.2.
 $(\text{LISTP } (\text{CONS } X1 \ X2))$

we get $(\text{LISTP } (\text{CONS } X \ Y))$ thus by instantiation of

Axiom 5.
 $X \neq F \rightarrow (\text{IF } X \ Y \ Z) = Y,$

and modus ponens we get

$$\begin{aligned} & (\text{IF } (\text{LISTP } (\text{CONS } X \ Y)) \\ & \quad (\text{CONS } (\text{CAR } (\text{CONS } X \ Y)) \\ & \quad \quad (\text{APPEND } (\text{CDR } (\text{CONS } X \ Y)) \ L)) \\ & \quad \quad L) \\ & \quad = \\ & (\text{CONS } (\text{CAR } (\text{CONS } X \ Y)) \\ & \quad \quad (\text{APPEND } (\text{CDR } (\text{CONS } X \ Y)) \ L)). \end{aligned}$$

Thus we have

$$\begin{aligned} & (\text{APPEND } (\text{CONS } X \ Y) \ L) \\ & \quad = \\ & (\text{CONS } (\text{CAR } (\text{CONS } X \ Y)) \\ & \quad \quad (\text{APPEND } (\text{CDR } (\text{CONS } X \ Y)) \ L)). \end{aligned}$$

It remains only to appeal to

Axiom 19.6.1.
 $(\text{EQUAL } (\text{CAR } (\text{CONS } X1 \ X2)) \ X1)$

and

Axiom 19.6.2.
 $(\text{EQUAL } (\text{CDR } (\text{CONS } X1 \ X2)) \ X2),$

to get

$$(\text{APPEND } (\text{CONS } X \ Y) \ L) = (\text{CONS } X \ (\text{APPEND } Y \ L)).$$

Q.E.D.

With **APPEND-NIL** and **APPEND-CONS** it is easy to derive

Theorem.

$$(\text{APPEND } \text{NIL} \ '(1 \ 2 \ 3)) = '(1 \ 2 \ 3)$$

Proof. The theorem is an instance of **APPEND-NIL**. Q.E.D.

Theorem.

$$(\text{APPEND } '(1 \ 2 \ 3) \ '(4 \ 5 \ 6)) = '(1 \ 2 \ 3 \ 4 \ 5 \ 6).$$

Proof.

$$\begin{aligned} & (\text{APPEND } '(1 \ 2 \ 3) \ '(4 \ 5 \ 6)) \\ &= (\text{CONS } 1 \ (\text{APPEND } '(2 \ 3) \ '(4 \ 5 \ 6))) \\ &= (\text{CONS } 1 \ (\text{CONS } 2 \ (\text{APPEND } '(3) \ '(4 \ 5 \ 6)))) \\ &= (\text{CONS } 1 \ (\text{CONS } 2 \ (\text{CONS } 3 \ (\text{APPEND } \text{NIL} \ '(4 \ 5 \ 6))))) \\ &= (\text{CONS } 1 \ (\text{CONS } 2 \ (\text{CONS } 3 \ '(4 \ 5 \ 6)))) \\ &= '(1 \ 2 \ 3 \ 4 \ 5 \ 6). \end{aligned}$$

The first three equalities are derived with **APPEND-CONS**, the fourth is derived with **APPEND-NIL**, and the last is an identity exploiting the **QUOTE** notation. Q.E.D.

5.4. The Associativity of APPEND

Theorem. ASSOCIATIVITY-OF-APPEND:

$$(\text{APPEND } (\text{APPEND } A \ B) \ C) = (\text{APPEND } A \ (\text{APPEND } B \ C)).$$

Proof. The proof is by induction on **A**. We split the problem into two cases according to whether **(LISTP A)**. The base case is when **A** is not a **LISTP**. The induction step is when **A** is a **LISTP**. In the induction step we assume one instance of the conjecture as an inductive hypothesis. The instance of interest is obtained by replacing **A** by **(CDR A)**. This induction is justified by the observation that **(COUNT A)** decreases under this substitution when **A** is a **LISTP**.

Having informally described the application of the induction principle we will now go through the statement and application of the principle carefully. Recall the principle:

Rule of Inference. Induction

Suppose

- (a) \mathbf{p} is a term;
- (b) \mathbf{m} is a term;
- (c) $\mathbf{q}_1, \dots, \mathbf{q}_k$ are terms;
- (d) $\mathbf{h}_1, \dots, \mathbf{h}_k$ are positive integers;
- (e) it is a theorem that (**ORDINALP** \mathbf{m});
and
- (f) for $1 \leq \mathbf{i} \leq \mathbf{k}$ and $1 \leq \mathbf{j} \leq \mathbf{h}_i$, $\mathbf{s}_{i,j}$
is a substitution and it is a theorem that

$$(\text{IMPLIES } \mathbf{q}_i \text{ (ORD-LESSP } \mathbf{m}/\mathbf{s}_{i,j} \mathbf{m})) .$$

Then \mathbf{p} is a theorem if

$$(1) (\text{IMPLIES (AND (NOT } \mathbf{q}_1) \dots (\text{NOT } \mathbf{q}_k)) \otimes \mathbf{T} \mathbf{p})$$

is a theorem and

for each $1 \leq \mathbf{i} \leq \mathbf{k}$,

$$(2) (\text{IMPLIES (AND } \mathbf{q}_i \mathbf{p}/\mathbf{s}_{i,1} \dots \mathbf{p}/\mathbf{s}_{i,h_i}) \otimes \mathbf{T} \mathbf{p})$$

is a theorem.

To use the principle of induction we must choose \mathbf{p} , the formula we wish to prove; \mathbf{m} , a “measure term” that justifies our induction; the terms \mathbf{q}_i that distinguish the various cases of the inductive proof; the substitutions $\mathbf{s}_{i,j}$ that produce the inductive hypotheses; and the various limits on the number of cases, substitutions, etc. In addition, we must prove certain theorems about our choices, namely that the measure term is an **ORDINALP** and that the substitutions decrease the measure under the appropriate tests. Here are the choices we shall make to prove the associativity of **APPEND**:

- (a) \mathbf{p} is the term (**EQUAL** (**APPEND** (**APPEND** \mathbf{A} \mathbf{B}) \mathbf{C}) (**APPEND** \mathbf{A} (**APPEND** \mathbf{B} \mathbf{C}))), the conjecture we are trying to prove;
- (b) \mathbf{m} , the measure, is (**COUNT** \mathbf{A});
- (c) \mathbf{k} , the number of cases, is 1 and the case, \mathbf{q}_1 , is (**LISTP** \mathbf{A});

- (d) h_1 , the number of induction hypotheses in the first case, is 1;
- (e) It is a theorem that $(\text{ORDINALP } (\text{COUNT } A))$, since, by **Axiom 13** **COUNT** returns a **NUMBERP** and from **Axiom 25** it follows that every **NUMBERP** is an **ORDINALP**; and
- (f) The substitution, $s_{1,1}$, that determines our induction hypothesis replaces A by $(\text{CDR } A)$. It is a theorem that

$(\text{IMPLIES } (\text{LISTP } A)$
 $\quad (\text{ORD-LESSP } (\text{COUNT } (\text{CDR } A))$
 $\quad \quad (\text{COUNT } A)))$.

An informal proof of the above theorem is as follows: by **Axiom 13**, **COUNT** returns a **NUMBERP** and thus, by **Axiom 24**, **ORD-LESSP** is **LESSP** on the arguments above. But by **Axiom 19.9**, the **COUNT** of A is one plus the sum of the **COUNTS** of $(\text{CAR } A)$ and $(\text{CDR } A)$. Hence, by simple arithmetic, the **COUNT** of $(\text{CDR } A)$ is **LESSP** the **COUNT** of A . The “simple arithmetic” needed is surprisingly elaborate if one starts at Ground Zero. In the next section we develop enough arithmetic to prove the above theorem more carefully. But for now we will continue with the proof of the associativity of **APPEND**.

Having met the requirements of the principle of induction, we may conclude that p is a theorem if

$$(1) \quad (\text{IMPLIES } (\text{AND } (\text{NOT } q_1) \dots (\text{NOT } q_k)) \otimes T_p)$$

is a theorem and, for each $1 \leq i \leq k$,

$$(2) \quad (\text{IMPLIES } (\text{AND } q_i \ p/s_{i,1} \dots p/s_{i,h_i}) \otimes T_p)$$

is a theorem. We will display and prove these formulas for the above choices of p , q_i , $s_{i,j}$, etc.

The base case is

$(\text{IMPLIES } (\text{NOT } (\text{LISTP } A))$
 $\quad (\text{EQUAL } (\text{APPEND } (\text{APPEND } A \ B) \ C)$
 $\quad \quad (\text{APPEND } A \ (\text{APPEND } B \ C))))$.

We will thus assume that A is not a **LISTP** and prove the conclusion above by successively rewriting both sides until they are identical. The only fact used in the rewriting is that **APPEND** returns its second argument when its first is not a

LISTP, an immediate observation from the definition of **APPEND** (Axiom 36). The desired conclusion is

$$\begin{aligned} &(\text{EQUAL } (\text{APPEND } (\text{APPEND } A \ B) \ C) \\ &\quad (\text{APPEND } A \ (\text{APPEND } B \ C))) \end{aligned}$$

By rewriting the second **APPEND** expression, this is equivalent to

$$\begin{aligned} &(\text{EQUAL } (\text{APPEND } B \ C) \\ &\quad (\text{APPEND } A \ (\text{APPEND } B \ C))), \end{aligned}$$

which, by rewriting the second **APPEND** expression, is equivalent to

$$\begin{aligned} &(\text{EQUAL } (\text{APPEND } B \ C) \\ &\quad (\text{APPEND } B \ C))). \end{aligned}$$

Thus, the base case holds.

Only one induction step is generated by the scheme above, since **k** is 1. The induction step is

$$\begin{aligned} &(\text{IMPLIES } (\text{AND } (\text{LISTP } A) \\ &\quad (\text{EQUAL } (\text{APPEND } (\text{APPEND } (\text{CDR } A) \ B) \\ &\quad \quad \quad C) \\ &\quad \quad (\text{APPEND } (\text{CDR } A) \\ &\quad \quad \quad (\text{APPEND } B \ C)))) \\ &\quad (\text{EQUAL } (\text{APPEND } (\text{APPEND } A \ B) \ C) \\ &\quad \quad (\text{APPEND } A \ (\text{APPEND } B \ C)))) \end{aligned}$$

We take **(LISTP A)** and

$$\begin{aligned} &(\text{EQUAL } (\text{APPEND } (\text{APPEND } (\text{CDR } A) \ B) \ C) \\ &\quad (\text{APPEND } (\text{CDR } A) \ (\text{APPEND } B \ C))) \end{aligned}$$

as hypotheses and will prove the conclusion above by successively rewriting it. The conclusion is

$$\begin{aligned} &(\text{EQUAL } (\text{APPEND } (\text{APPEND } A \ B) \\ &\quad \quad \quad C) \\ &\quad (\text{APPEND } A \\ &\quad \quad (\text{APPEND } B \ C))). \end{aligned}$$

By the definition of **APPEND** and the fact that **A** is a **LISTP** we can rewrite **(APPEND A B)** to **(CONS (CAR A) (APPEND (CDR A) B))**. Thus, the conclusion becomes

$$\begin{aligned} &(\text{EQUAL } (\text{APPEND } (\text{CONS } (\text{CAR } A) \\ &\quad \quad \quad (\text{APPEND } (\text{CDR } A) \ B)) \\ &\quad \quad \quad C) \\ &\quad (\text{APPEND } A \ (\text{APPEND } B \ C))). \end{aligned}$$

By **APPEND-CONS** the first **APPEND** expression above can be rewritten, producing

```
(EQUAL (CONS (CAR A)
              (APPEND (APPEND (CDR A) B)
                      C))
      (APPEND A
              (APPEND B C))).
```

By the definition of **APPEND** and the hypothesis that **A** is a **LISTP** we can rewrite the third **APPEND** expression above to produce

```
(EQUAL (CONS (CAR A)
              (APPEND (APPEND (CDR A) B)
                      C))
      (CONS (CAR A)
            (APPEND (CDR A)
                    (APPEND B C)))).
```

By our induction hypothesis we can replace

```
(APPEND (APPEND (CDR A) B)
      C)
```

above by

```
(APPEND (CDR A)
      (APPEND B C)).
```

Thus the conclusion becomes

```
(EQUAL (CONS (CAR A)
              (APPEND (CDR A)
                      (APPEND B C)))
      (CONS (CAR A)
            (APPEND (CDR A)
                    (APPEND B C)))).
```

which is an identity. Thus the induction step holds. Q.E.D.

5.5. Simple Arithmetic

In the previous section we performed an induction in which we assumed the formula for **(CDR A)** and proved it for **A**, under the hypothesis **(LISTP A)**. This induction was justified by the observation that the **COUNT** of **(CDR A)** is smaller than that of **A** when **A** is a **LISTP**. In general, if **r** is the recognizer of a

shell class with base object (**base**) and accessors $\mathbf{ac}_1, \dots, \mathbf{ac}_n$, it is a theorem that

$$\begin{aligned} &(\text{IMPLIES } (\text{AND } (\mathbf{r} \ \mathbf{X}) \\ &\quad (\text{NOT } (\text{EQUAL } \mathbf{X} \ (\mathbf{base})))) \\ &\quad (\text{ORD-LESSP } (\text{COUNT } (\mathbf{ac}_i \ \mathbf{X})) \ (\text{COUNT } \mathbf{X}))) \end{aligned}$$

for each $1 \leq i \leq n$. This permits “structural” inductions on the shell class **r**.

In this section we prove the simple arithmetic theorems leading to this result. In the next section we prove the above metatheorem and explain what we mean by “structural inductions.”

Theorem. NUMBERP-SUB1:
(NUMBERP (SUB1 X))

Proof. The proof is by cases. If **X** is not a **NUMBERP** or is 0, (SUB1 X) is 0 by **Axiom 12.7**, and 0 is a **NUMBERP** by **Axiom 12.3**. Thus, assume **X** is a non-0 **NUMBERP**. By **Axiom 12.5**, **X** is (ADD1 (SUB1 X)). If (SUB1 X) is a **NUMBERP** we are done. Thus assume the contrary. Then by **Axiom 12.7**, (SUB1 (ADD1 (SUB1 X))) is 0 and hence a **NUMBERP**. Q.E.D.

Theorem. NUMBERP-PLUS:
(NUMBERP (PLUS I J))

Proof. By the definition of **PLUS** (**Axiom 18**), (PLUS I J) is either (FIX J) or the result of an **ADD1**. If the former, it is a **NUMBERP** by the definition of **FIX** (**Axiom 17**). If the latter, it is a **NUMBERP** by **Axiom 12.2**. Q.E.D.

Theorem. ASSOCIATIVITY-OF-PLUS:
(EQUAL (PLUS (PLUS I J) K)
 (PLUS I (PLUS J K))).

Proof. We will induct on **I**. The case split is on (NOT (ZEROP I)). In the induction step we will assume the instance obtained by replacing **I** by (SUB1 I). The measure we use is (FIX I).

The formulas we must prove to justify the induction are

$$(\text{ORDINALP } (\text{FIX } \mathbf{I}))$$

and

$$\begin{aligned} &(\text{IMPLIES } (\text{NOT } (\text{ZEROP } \mathbf{I})) \\ &\quad (\text{ORD-LESSP } (\text{FIX } (\text{SUB1 } \mathbf{I})) \ (\text{FIX } \mathbf{I}))) \end{aligned}$$

The first theorem is trivial since **FIX** always returns a **NUMBERP** (by case analysis on (NUMBERP I)).

The second theorem is more interesting. By **NUMBERP-SUB1** we know that (SUB1 I) is a **NUMBERP**. Hence, (FIX (SUB1 I)) is (SUB1 I). By

the hypothesis that **I** is non-**ZEROP** and the definition of **ZEROP** (**Axiom 16**) we know **I** is a **NUMBERP**. Hence, **(FIX I)** is **I**. Thus, the second theorem can be simplified to

$$(\text{IMPLIES } (\text{NOT } (\text{ZEROP } I)) \\ (\text{ORD-LESSP } (\text{SUB1 } I) I)).$$

By the definition of **ORD-LESSP** (**Axiom 24**), **ORD-LESSP** is **LESSP** on the arguments above, and thus the formula becomes

$$(\text{IMPLIES } (\text{NOT } (\text{ZEROP } I)) \\ (\text{LESSP } (\text{SUB1 } I) I))$$

But this is just **Axiom 23**.

In short, an induction in which we replace **I** by **(SUB1 I)** when **I** is a non-**ZEROP** is justified because **Axiom 23** says it is.

We now continue with the proof of the associativity of **PLUS**.

The base case for the associativity of **PLUS** is

$$(\text{IMPLIES } (\text{NOT } (\text{NOT } (\text{ZEROP } I)))) \\ (\text{EQUAL } (\text{PLUS } (\text{PLUS } I J) K) \\ (\text{PLUS } I (\text{PLUS } J K)))).$$

By the definitions of **NOT** (**Axiom 8**) and **ZEROP** (**Axiom 16**) and propositional calculus, the hypothesis is equivalent to **(ZEROP I)**. Under that hypothesis, the conclusion is equal to

$$(\text{EQUAL } (\text{PLUS } (\text{FIX } J) K) \\ (\text{FIX } (\text{PLUS } J K)))$$

by the definition of **PLUS** (**Axiom 18**). By case analysis on whether **J** is a **NUMBERP** and the definitions of **PLUS**, **ZEROP**, and **FIX**, both sides above are equal to **(PLUS J K)**.

The induction step for the associativity of **PLUS** is

$$(\text{IMPLIES } (\text{AND } (\text{NOT } (\text{ZEROP } I)) \\ (\text{EQUAL } (\text{PLUS } (\text{PLUS } (\text{SUB1 } I) J) K) \\ (\text{PLUS } (\text{SUB1 } I) (\text{PLUS } J K)))) \\ (\text{EQUAL } (\text{PLUS } (\text{PLUS } I J) K) \\ (\text{PLUS } I (\text{PLUS } J K)))).$$

Under the hypothesis that **I** is non-**ZEROP**, the definition of **PLUS** can be used to rewrite the second and third calls of **PLUS** in the conclusion above to produce the new goal:

$$(\text{EQUAL } (\text{PLUS } (\text{ADD1 } (\text{PLUS } (\text{SUB1 } I) J)) K) \\ (\text{ADD1 } (\text{PLUS } (\text{SUB1 } I) (\text{PLUS } J K)))).$$

By the definition of **ZEROP** and **Axioms 12.2** and **12.4** we can show that no **ADD1**-expression is **ZEROP**, and thus we can apply the definition of **PLUS** to the first call of **PLUS** above and produce

```
(EQUAL
  (ADD1 (PLUS (SUB1 (ADD1 (PLUS (SUB1 I) J))) K))
  (ADD1 (PLUS (SUB1 I) (PLUS J K)))).
```

We can use **Axiom 12.6** and **NUMBERP-PLUS** to simplify the **(SUB1 (ADD1 (PLUS ...)))** expression above and produce

```
(EQUAL (ADD1 (PLUS (PLUS (SUB1 I) J) K))
  (ADD1 (PLUS (SUB1 I) (PLUS J K)))).
```

Finally, substituting the right-hand side of the induction hypothesis for the left into the conclusion above we produce the identity

```
(EQUAL (ADD1 (PLUS (SUB1 I) (PLUS J K)))
  (ADD1 (PLUS (SUB1 I) (PLUS J K)))).
```

Q.E.D.

Using similar techniques we can prove the following theorems:

Theorem. PLUS-RIGHT-ID:
(IMPLIES (ZEROP J)
 (EQUAL (PLUS I J) (FIX I)))

Theorem. COMMUTATIVITY-OF-PLUS:
(EQUAL (PLUS I J) (PLUS J I))

More interesting is the proof of the transitivity of **LESSP**:

Theorem. TRANSITIVITY-OF-LESSP:
(IMPLIES (AND (LESSP I J)
 (LESSP J K))
 (LESSP I K)).

Proof. We will induct on **I**, using the case split

```
(AND (NOT (ZEROP I))
  (NOT (ZEROP J))
  (NOT (ZEROP K))).
```

In the induction step we will have one induction hypothesis obtained by replacing **I** by **(SUB1 I)**, **J** by **(SUB1 J)** and **K** by **(SUB1 K)**. This substitution decreases the **FIX** of **I** just as in the previous example.

The base case reduces, with propositional calculus, to three subcases, one for each of the assumptions that **I**, **J**, or **K** is **ZEROP**. In each subcase we must prove

```
(IMPLIES (AND (LESSP I J)
              (LESSP J K))
          (LESSP I K))
```

In the two cases when either **J** or **K** is **ZEROP**, the definition of **LESSP** (Axiom 22) can be used to reduce one of the two **LESSP** hypotheses to **F**, and propositional calculus immediately completes the proof of the subcases. If **I** is **ZEROP**, (**LESSP I J**) is equivalent (by the definitions of **LESSP** and **NOT**) to (**NOT (ZEROP J)**), and (**LESSP I K**) is (**NOT (ZEROP K)**). Thus we wish to prove

```
(IMPLIES (AND (NOT (ZEROP J))
              (LESSP J K))
          (NOT (ZEROP K))),
```

which, by propositional calculus, is equivalent to

```
(IMPLIES (AND (NOT (ZEROP J))
              (ZEROP K))
          (NOT (LESSP J K))).
```

But by the definition of **LESSP**, if **K** is **ZEROP**, (**LESSP J K**) is **F**, and propositional calculus finishes the subcase.

The induction step for the transitivity of **LESSP** is

```
(IMPLIES (AND (NOT (ZEROP I))
              (NOT (ZEROP J))
              (NOT (ZEROP K))
              (IMPLIES (AND (LESSP (SUB1 I)
                                (SUB1 J))
                          (LESSP (SUB1 J)
                                (SUB1 K))))
              (LESSP (SUB1 I)
                      (SUB1 K))))
          (IMPLIES (AND (LESSP I J)
                      (LESSP J K))
                  (LESSP I K)))
```

But under the three non-**ZEROP** assumptions, the conclusion

```
(IMPLIES (AND (LESSP I J)
              (LESSP J K))
          (LESSP I K))
```

can be rewritten, by expanding all three calls of **LESSP**, to

```
(IMPLIES (AND (LESSP (SUB1 I)
                     (SUB1 J))
              (LESSP (SUB1 J)
                     (SUB1 K)))
          (LESSP (SUB1 I)
                 (SUB1 K)))
```

which is the induction hypothesis. Q.E.D.

We can also prove such results as

Theorem. LESSP-PLUS:
 (NOT (LESSP (PLUS I J) I))

Theorem. MIXED-TRANSITIVITY-OF-LESSP:
 (IMPLIES (AND (LESSP I J)
 (NOT (LESSP K J)))
 (LESSP I K))

Intuitively, (NOT (LESSP x y)) is $y \leq x$. The first theorem above is $I \leq I+J$, where $+$ is on the natural numbers only. The second theorem is $I < J \wedge J \leq K \rightarrow I < K$.

5.6. Structural Induction

We wish to prove, for every shell recognizer r and its corresponding base function symbol **base** and each of its n accessor functions ac_i :

```
(IMPLIES (AND (r X)
              (NOT (EQUAL X (base))))
          (ORD-LESSP (COUNT (aci X)) (COUNT X)))
```

Suppose we had such a theorem. Then it would be possible to perform any induction on X in which the case split is on

```
(AND (r X)
      (NOT (EQUAL X (base))))
```

and n induction hypotheses are provided, each obtained by replacing X by $(ac_i X)$ for some $1 \leq i \leq n$. Such an induction would be justified by **COUNT** by appeal to the theorem schema above.

We call such an induction a *structural induction for the shell type r* because it inductively decomposes structures of type r into their substructures until “bot-

coming out'' on non-**r** objects or (**base**). Many beautiful examples of structural induction are given by Burstall, who coined the term, in [44]. Our proof of the associativity of **APPEND** is an example of a structural induction: The base case was when (**NOT** (**LISTP** **A**)). The induction step was when (**LISTP** **A**), and we inductively assumed the conjecture for **A** replaced by (**CDR** **A**). We could have also assumed the conjecture for (**CAR** **A**). The familiar induction on **I** by (**SUB1** **I**) is a structural induction of type **NUMBERP**.

We now prove the justifying theorem schema. Familiarity with the shell principle is required to follow the proof.

Metatheorem. *Structural Induction Justification:*

For every shell recognizer, **r**, with corresponding base function symbol **base**, and for each of the **n** accessor function symbols **ac_i**, $1 \leq i \leq n$

```
(IMPLIES (AND (r X)
               (NOT (EQUAL X (base))))
          (ORD-LESSP (COUNT (aci X)) (COUNT X))).
```

Proof. By **Axiom 13**, **COUNT** returns a **NUMBERP** and hence by the definition of **ORD-LESSP** (**Axiom 24**), the **ORD-LESSP** call above can be replaced by a **LESSP** call. By the ninth shell axiom schema we know that when **X** is a non-(**base**) object of type **r**, the **COUNT** of **X** is one greater than the sum of the **COUNT**s of its components. Replacing (**COUNT** **X**) by the right-hand side of the ninth shell axiom schema renders our conclusion:

```
(LESSP (COUNT (aci X))
        (ADD1 (PLUS (COUNT (ac1 X))
                    ...
                    (COUNT (aci X))
                    ...
                    (COUNT (acn X))))))
```

By repeated applications of **ASSOCIATIVITY-OF-PLUS** and **COMMUTATIVITY-OF-PLUS** our goal becomes

```
(LESSP (COUNT (aci X))
        (ADD1 (PLUS (COUNT (aci X))
                    (PLUS (COUNT (ac1 X))
                        ...
                        (COUNT (aci-1 X))
                        (COUNT (aci+1 X))
                        ...
                        (COUNT (acn X)))))))
```

We now expand the definition of **LESSP** (**Axiom 22**) and consider the cases on

whether the first argument above, $(\text{COUNT } (\text{ac}_i \text{ X}))$, is **ZEROP**. If the first argument is **ZEROP** the goal is reduced to **T** since the second argument is not **ZEROP** by **Axioms 12.2** and **12.4**. If the first argument is not **ZEROP** the recursive case obtains and, using **Axiom 12.6** and **NUMBERP-PLUS**, the goal becomes

$$\begin{aligned} &(\text{LESSP } (\text{SUB1 } (\text{COUNT } (\text{ac}_i \text{ X}))) \\ &\quad (\text{PLUS } (\text{COUNT } (\text{ac}_i \text{ X})) \\ &\quad\quad (\text{PLUS } (\text{COUNT } (\text{ac}_1 \text{ X})) \\ &\quad\quad\quad \vdots \\ &\quad\quad\quad (\text{COUNT } (\text{ac}_n \text{ X})))))) \end{aligned}$$

Observe that this goal has the form $(\text{LESSP } (\text{SUB1 } i) (\text{PLUS } i \ j))$ where we know that i is not **ZEROP**. But by **Axiom 23** we have that $(\text{LESSP } (\text{SUB1 } i) \ i)$, and by **LESSP-PLUS** we have that $(\text{NOT } (\text{LESSP } (\text{PLUS } i \ j) \ i))$. Hence, by **MIXED-TRANSITIVITY-OF-LESSP**, we have the goal $(\text{LESSP } (\text{SUB1 } i) (\text{PLUS } i \ j))$. Q.E.D.

We can strengthen our position somewhat more by observing

Metatheorem. LESSP-COUNT-ac_i :
 $(\text{NOT } (\text{LESSP } (\text{COUNT } X) (\text{COUNT } (\text{ac}_i \text{ X}))))$

Proof Sketch. If X is a non-base object of the correct type the **COUNT** of $(\text{ac}_i \text{ X})$ is smaller than that of X ; otherwise, $(\text{ac}_i \text{ X})$ is the default value for the i th accessor, which is some base object, and hence has **COUNT** 0. Q.E.D.

By combining this result with **MIXED-TRANSITIVITY-OF-LESSP** we can conclude that

Metatheorem. *Generalized Structural Induction Justification:*

For every shell recognizer, r , with corresponding base function symbol **base**

$$\begin{aligned} &(\text{IMPLIES } (\text{AND } (r \text{ X}) \\ &\quad (\text{NOT } (\text{EQUAL } X (\text{base})))) \\ &\quad (\text{ORD-LESSP } (\text{COUNT } \text{nest}) (\text{COUNT } X))) \end{aligned}$$

where **nest** is any nest of corresponding shell accessors around X .

For example if I is a non-0 **NUMBERP** then the **COUNT** of $(\text{SUB1 } (\text{SUB1 } I))$ is **ORD-LESSP** the **COUNT** of I . Similarly, $(\text{CADDAR } X)$ has a smaller **COUNT** than X if X is a **LISTP** (whether or not X has as much **LISTP** substructure as suggested by **CADDAR**).

Henceforth, we will call a *structural induction* any induction that splits on whether the induction variable is a non-base object of a given type and then provides as its inductive hypotheses instances of the conjecture obtained by

replacing the induction variable by arbitrary nests of shell accessors applied to the induction variable.

Part II

Using the System

6 Mechanized Proofs in the Logic

In this chapter we discuss the mechanization of proofs in this logic. In particular, we describe at a very high level how our mechanical theorem prover works, and we show a proof produced by it.

6.1. The Organization of Our Theorem Prover

In the last chapter we saw that it is possible to derive new rules of inference from the axioms and rules given, e.g., structural induction. Any proof using structural induction can be converted into a proof within the given formal system. Logicians and mathematicians are very fond of derived rules of inference because they permit the formal logic to be relatively primitive while allowing the production of sophisticated proofs. A well-known and very useful derived rule of inference is the tautology theorem: a formula of propositional calculus has a proof if and only if it is valid under a truth table. More complicated derived rules are those that justify the use of equality decision procedures and certain arithmetic decision procedures.

Our mechanical theorem prover uses a variety of such high-level derived rules of inference to discover and describe its proofs. In particular it uses the following six proof techniques:

- **Simplification:** a combination of decision procedures (for propositional calculus, equality, and linear arithmetic), term rewriting, and “metafunctions,” which are user-supplied simplifiers that have been mechanically proved correct;
- **Destructor Elimination:** the trading of “bad” terms for “good” by choosing an appropriate representation for the objects being manipulated;
- **Cross-fertilization:** the heuristic use and elimination of equality hypotheses;
- **Generalization:** the adoption of a more general goal obtained by replacing terms in the given goal by new variables;
- **Elimination of Irrelevance:** the discarding of apparently unimportant hypotheses; and
- **Induction.**

As implemented, each of these proof techniques is a computer program that takes a formula as input and yields a set of formulas as output; the input formula is provable if each of the output formulas is. We call each of these six programs a “process.” From the logical point of view, each such process is a derived rule of inference that is “run backwards.” From the problem solving point of view, each process “reduces” the input goal to a conjunction of subgoals.

Not every process is applicable to every formula. For example, it may not be possible to further simplify a formula. When a process is applied to an inappropriate formula, the process returns the singleton set containing the formula, i.e., it implements the trivial derived rule that p can be proved if p can be proved. For example, the simplifier returns the singleton set containing its input formula if that input formula is already in simplest form.

Sometimes a process recognizes that its input is a theorem. For example, tautologies are recognized by the simplifier. In this case the output set is the empty set.

The theorem prover is organized around a “pool” of goal formulas. Initially the user places an input conjecture into the pool. Formulas are drawn out of the pool one at a time for consideration. Each formula is passed in turn to the six processes in the order they are listed above until some process is applicable.¹⁰ When an applicable process produces a new set of subgoals, each is added to the

¹⁰Roughly speaking, a formula is printed when its offspring are deposited into the pool. By waiting until that point, we can explain which process hit the formula, what was done to it, and how many new goals have been generated. Of course, special cases arise when a formula is proved without the generation of new goals, etc.

pool. The consideration of goals in the pool continues until either the pool is empty and no new subgoals are produced—in which case the system has “won”—or one of several termination conditions is met—in which case the system has “lost.” The system may “run forever” (until the host system’s resources are exhausted), though this happens only rarely.

Organization of the Theorem Prover

In particular, when the user puts a formula into the pool, it is first drawn out by the simplifier. If the simplifier applies, it puts into the pool a set of new formulas whose conjunction is sufficient to prove the extracted formula. If the simplifier does not apply (i.e., the formula cannot be further simplified), the formula is passed to the destructor elimination process. If destructor elimination is applicable, it puts a new set of formulas into the pool; if not, it passes the formula to the cross-fertilization process, etc. This continues all the way around the circle of processes.

Roughly speaking, the first two processes in the circle preserve equivalence and are concerned primarily with simplifying formulas. The next three processes strengthen the formula in anticipation of the use of induction. By the time a formula arrives at the induction process it has been simplified and generalized as much as possible by our techniques.

When the system wins—i.e., the pool is finally emptied—the trace of the theorem prover is a proof of the input conjecture, provided each of the six processes is considered a derived rule of inference. When the system loses, the user should infer nothing about the validity of the original conjecture; the initial problem may or may not be a theorem. Even when the system quits on a

subgoal that is manifestly invalid the original conjecture may be a theorem that was “lost” by overgeneralization in one of the processes.

Finally, the behavior of three of these processes, namely simplification, destructor elimination, and generalization, can be heavily influenced by rules derived from previously proved, user-supplied theorems. As we have mentioned, having an appropriate data base of rules is crucial to the machine’s proof of deep theorems. We will deal at length with this issue later.

6.2. An Example Proof—Associativity of TIMES

Here is a proof produced by the theorem prover. We prove that multiplication is associative. All of the text in **typewriter font** was produced by the theorem prover. We have inserted the labels *1, *2, ..., in the proof so that we can discuss the proof in the next section. When the proof begins, the theorem prover is in the “Ground Zero” state: its data base of rules contains only those corresponding to the axioms and definitions in Chapter 4.

Theorem.

```
(EQUAL (TIMES (TIMES I J) K)
        (TIMES I (TIMES J K)))
```

Proof.

Call the conjecture *1. *1

Perhaps we can prove it by induction. Three inductions are suggested by terms in the conjecture. They merge into two likely candidate inductions. However, only one is unflawed. We will induct according to the following scheme:

```
(AND
  (IMPLIES (ZEROP I) (p I J K))
  (IMPLIES (AND (NOT (ZEROP I)) (p (SUB1 I) J K))
            (p I J K))). *2
```

Linear arithmetic, the lemma COUNT-NUMBERP, and the definition of ZEROP can be used to prove that the measure (COUNT I) decreases according to the well-founded relation LESSP in each induction step of the scheme. The above induction scheme leads to two new goals:

```
Case 2. (IMPLIES (ZEROP I)
                 (EQUAL (TIMES (TIMES I J) K)
                        (TIMES I J K))), *3
```

which simplifies, opening up the definitions of ZEROP, EQUAL, and TIMES, to:

T.

```
Case 1. (IMPLIES                                     *4
  (AND (NOT (ZEROP I))
    (EQUAL (TIMES (TIMES (SUB1 I) J) K)
      (TIMES (SUB1 I) J K)))
  (EQUAL (TIMES (TIMES I J) K)
    (TIMES I J K))),
```

which simplifies, expanding ZEROP and TIMES, to:

```
(IMPLIES                                             *5
  (AND (NOT (EQUAL I 0))
    (NUMBERP I)
    (EQUAL (TIMES (TIMES (SUB1 I) J) K)
      (TIMES (SUB1 I) J K)))
  (EQUAL (TIMES (PLUS J (TIMES (SUB1 I) J)) K)
    (PLUS (TIMES J K)
      (TIMES (SUB1 I) J K)))).
```

Appealing to the lemma SUB1-ELIM, we now replace I by (ADD1 X) to eliminate (SUB1 I). We rely upon the type restriction lemma noted when SUB1 was introduced to constrain the new variable. This generates:

```
(IMPLIES                                             *6
  (AND (NUMBERP X)
    (NOT (EQUAL (ADD1 X) 0))
    (EQUAL (TIMES (TIMES X J) K)
      (TIMES X J K)))
  (EQUAL (TIMES (PLUS J (TIMES X J)) K)
    (PLUS (TIMES J K) (TIMES X J K)))).
```

This further simplifies, trivially, to:

```
(IMPLIES                                             *7
  (AND (NUMBERP X)
    (EQUAL (TIMES (TIMES X J) K)
      (TIMES X J K)))
  (EQUAL (TIMES (PLUS J (TIMES X J)) K)
    (PLUS (TIMES J K) (TIMES X J K)))).
```

We use the above equality hypothesis by substituting (TIMES (TIMES X J) K) for (TIMES X J K) and throwing away the equality. This generates the goal:

```

(IMPLIES (NUMBERP X)                                     *8
  (EQUAL (TIMES (PLUS J (TIMES X J)) K)
    (PLUS (TIMES J K)
      (TIMES (TIMES X J) K))))).

```

We will try to prove the above formula by generalizing it, replacing (TIMES X J) by Y. We restrict the new variable by recalling the type restriction lemma noted when TIMES was introduced. We thus obtain:

```

(IMPLIES                                               *9
  (AND (NUMBERP Y) (NUMBERP X))
  (EQUAL (TIMES (PLUS J Y) K)
    (PLUS (TIMES J K) (TIMES Y K))))),

```

which has an irrelevant term in it. By eliminating the term we get:

```

(IMPLIES                                               *10
  (NUMBERP Y)
  (EQUAL (TIMES (PLUS J Y) K)
    (PLUS (TIMES J K) (TIMES Y K))))),

```

which we will finally name *1.1. *11

Perhaps we can prove it by induction. Three inductions are suggested by terms in the conjecture. They merge into two likely candidate inductions. However, only one is unflawed. We will induct according to the following scheme: *12

```

(AND
  (IMPLIES (ZEROP J) (p J Y K))
  (IMPLIES (AND (NOT (ZEROP J)) (p (SUB1 J) Y K))
    (p J Y K))).

```

Linear arithmetic, the lemmas SUB1-LESSEQP and SUB1-LESSP, and the definition of ZEROP establish that the measure (COUNT J) decreases according to the well-founded relation LESSP in each induction step of the scheme. The above induction scheme leads to the following two new formulas:

Case 2. (IMPLIES
 (AND (ZEROP J) (NUMBERP Y))
 (EQUAL (TIMES (PLUS J Y) K)
 (PLUS (TIMES J K) (TIMES Y K))))).

This simplifies, unfolding ZEROP, EQUAL, PLUS, and TIMES, to:

T.

Case 1. (IMPLIES *13
 (AND
 (NOT (ZEROP J))
 (EQUAL (TIMES (PLUS (SUB1 J) Y) K)
 (PLUS (TIMES (SUB1 J) K) (TIMES Y K)))
 (NUMBERP Y))
 (EQUAL (TIMES (PLUS J Y) K)
 (PLUS (TIMES J K) (TIMES Y K)))).

This simplifies, applying SUB1-ADD1, and opening up the functions ZEROP, PLUS, and TIMES, to the formula:

```
(IMPLIES
 (AND
  (NOT (EQUAL J 0))
  (NUMBERP J)
  (EQUAL (TIMES (PLUS (SUB1 J) Y) K)
   (PLUS (TIMES (SUB1 J) K) (TIMES Y K)))
  (NUMBERP Y))
 (EQUAL (PLUS K (TIMES (PLUS (SUB1 J) Y) K))
  (PLUS (PLUS K (TIMES (SUB1 J) K))
   (TIMES Y K)))).
```

This again simplifies, using linear arithmetic, to:

T.

That finishes the proof of *1.1, which finishes *14
 the proof of *1. Q.E.D.

6.3. An Explanation of the Proof

Between the time the conjecture was submitted and the time the system printed the text at *1, each of the first five processes attempted to transform the formula and none succeeded. At *1 the induction process got the formula and chose to induct on **I**. The paragraph below *1 explains why this induction was chosen. The implementation of our induction mechanism is described in detail in [24] where we define such notions as the inductions “suggested” by a term, the “merge” of two inductions, and when an induction is “flawed.” The induction schema chosen is given in *2, with (p **I** **J** **K**) used schematically to

represent the entire conjecture. Note that the induction is structural induction of type **NUMBERP** on **I**. The paragraph following *2 explains why this induction is permitted under our Principle of Induction. The induction process puts two new formulas into the pool, the base case and the induction step.

Formula *3 is the base case. The theorem prover labels the formula **Case 2**.¹¹ The base case is drawn out of the pool by the simplifier and is reduced to **T**. Formula *4 is the induction step. It is drawn out of the pool next and is simplified to *5. Technically, *5 is put back into the pool, drawn out again by the simplifier, and cannot be further simplified. Formula *5 is thus handed off to the next process, destructor elimination.

Because the formula has (**SUB1 I**) in it, the destructor elimination process chooses to replace **I** by (**ADD1 X**), where **X** is a new, numerically typed variable. This eliminates (**SUB1 I**), which is now **X**. The output of the destructor elimination process is *6, which is put back into the pool and resimplified.

Formula *6 has a trivial hypothesis, namely the second, which says (**ADD1 X**) is not **0**, a remnant of the first hypothesis of *5 which said **I** was not **0** and which is now obviously true by the representation chosen for **I**. The simplifier removes this hypothesis without using any named rules. That is why it says formula *6 *trivially* simplifies to *7.

Formula *7 is put into the pool and is subjected to both simplification and destructor elimination without effect before it arrives at the cross-fertilization process. Here the equality hypothesis, which is actually the induction hypothesis, is substituted into the simplified induction conclusion. Furthermore, the induction hypothesis is then discarded on the heuristic grounds that it has been “used.” Because a hypothesis is discarded, the output of the cross-fertilization process may not be a theorem even though the input is. All that is guaranteed is that if the output is a theorem, then the input can be proved from it. The output is *8, which is passed through all the preceding processes and eventually arrives at the generalization process.

The generalizer notes that (**TIMES X J**) occurs on both the left- and right-hand sides of the induction conclusion and decides to attack the more general conjecture obtained by replacing (**TIMES X J**) by the new variable **Y**. Because (**TIMES X J**) is known to be a number, the generalizer restricts **Y** to being a number. The output of the generalizer is *9. Like cross-fertilization, the generalizer may produce a non-theorem as its output even if its input is a theorem.

¹¹In general the system numbers cases backwards. The reason for this is so that when you return to your terminal with your coffee and find it working on **Case 57** you know how far you have to go.

Formula *9, after traversal through all of the preceding processes, arrives at the process that eliminates irrelevance. The hypothesis that \mathbf{x} is a number is judged unnecessary and discarded because \mathbf{x} does not occur elsewhere in the conjecture. The output is *10. In fact, the other hypothesis, (**NUMBERP** \mathbf{Y}) is also unnecessary, but the theorem prover does not detect that.

Formula *10 is put in the pool and cycles through all of the first five processes untouched. The system will attack it by induction. However, rather than start on the induction immediately, we prefer to finish processing any remaining formulas in the pool. Thus, at *11 we generate a name for the formula, *1.1, and set it aside to be proved by induction.

It is worth noting that the formula *1.1 is in fact the statement that multiplication distributes over addition. It was generated automatically by the combination of the foregoing heuristics and adopted as a subgoal to prove the associativity of multiplication.

After naming *1.1 and setting the formula aside, the system continues processing formulas in the pool. However, there are none in this example. Thus, having finished with all the cases of the last induction, the system picks up the formula set aside and at *12 applies the induction process to it.

The induction process chooses the obvious induction on \mathbf{J} and produces two new cases, each of which is proved by simplification. Note that the induction step, *13, requires two applications of simplification, the second using the linear arithmetic decision procedure.

Finally, at *14, the system notes that all formulas have been proved.

7

An Introduction to the System

Of course, it takes more to mechanize our logic than merely a mechanical theorem prover. The system widely known as the “Boyer-Moore theorem prover” includes the theorem proving program sketched above but also provides a large number of features having little to do with theorem proving, such as a mechanization of the shell and definition principles, dependency tracking, a compiler for the logic, a run-time environment for executing functions in the logic, and various facilities for keeping track of the evolving data base of rules.

This chapter is essentially a tutorial introduction to our system. We introduce the notion of the data base and its rules, the logical “events” that add new rules to the data base, the most commonly used commands, and various issues of general concern such as errors, aborting computations, etc.

7.1. The Data Base of Rules

The theorem proving system is, in essence, a mechanical tool for building theories. The user generally constructs his or her new theory from an old one by adding shell declarations and function definitions and proving theorems about the new concepts. The system is organized as an interactive command interpreter that performs operations on a global data base representing the evolving theory.

The data base contains the axioms, the user-defined shells, defined functions, and theorems proved by the system. Some commands add facts (e.g., definitions or proved theorems) to the data base. Other commands permit the user to inspect the data base. Still other commands remove facts from the data base. When facts are removed, logically dependent facts are also removed. Finally, commands are provided for writing the data base to a file and restoring it from a file so that work can be saved from one session to the next.

In a shallow sense, the theorem prover is fully automatic: the system requires no advice or directives from the user once a proof attempt has started. The only way the user can alter the behavior of the system during a proof attempt is to abort the proof attempt. However, in a deeper sense, the theorem prover is interactive: the data base—and hence the user's past actions—influences the behavior of the theorem prover. Five of the proof techniques—namely linear arithmetic, rewriting, metafunctions, destructor elimination, and generalization—use the data base as a source of *rules* to apply to the conjecture at hand.

From where do these rules come? They are generated from the theorems formulated by the user and previously proved by the system. When a theorem is submitted for proof, the user attaches to it one or more tokens that declare what classes of rules are to be generated from the theorem, if proved. The tokens are taken from the set {**REWRITE META ELIM GENERALIZE**}. Given a class of rule to be generated, e.g., **REWRITE**, the precise form of the rule generated from a theorem is dependent upon the syntactic form of theorem.

For example, the theorem

```
(IMPLIES (NOT (MEMBER X A))
          (EQUAL (DELETE X A) A))
```

generates the **REWRITE** rule: Replace (instances of) **(DELETE X A)** by (the instance of) **A** provided (the instance of) **(MEMBER X A)** rewrites to **F**. The equivalent formula

```
(IMPLIES (NOT (EQUAL (DELETE X A) A))
          (MEMBER X A))
```

generates a radically different rule: Replace **(MEMBER X A)** by **T** provided **(EQUAL (DELETE X A) A)** rewrites to **F**.

Propositionally equivalent theorems may have radically different interpretations as rules. Even the order in which theorems are proved is important since it affects the order in which the corresponding rules are applied.

Each rule has an associated *status* indicating whether the rule is *enabled* or *disabled*. The status of a rule can be set by the user. When a rule is disabled it is never applied.¹²

¹²Well, almost never! For example, we do not check the status of rules stored as type prescriptions.

A data base is thus more than a logical theory: it is a set of rules for proving theorems in the given theory. The user leads the theorem prover to “difficult” proofs by “programming” its rule base. Given a goal theorem, the user generally discovers a proof himself, identifies the key steps in the proof, and then formulates them as lemmas, paying particular attention to their interpretation as rules.

The key role of the user of our system is guiding the theorem prover to proofs by the strategic selection of the sequence of theorems to prove and the proper formulation of those theorems. Successful users of the system must know how to prove theorems in the logic and must understand how the theorem prover interprets them as rules.

7.2. Logical Events

The act of adding a new rule to the data base or changing the status of a rule in the data base is called an *event*. Some events are associated with an extension of the logic itself, e.g., the addition of a new axiom (and its generated rules), the addition of a new shell (and its generated rules) or the addition of a new function definition (and its generated rules). Other events are associated merely with an extension of the syntax, namely the declaration of the arity of a new function symbol. The most common event is the certification of a formula as a theorem (and the addition of its generated rules).

Each event has a *name*, which is a word chosen by the user, and generally is either the name of a function introduced by the event or the name of the axiom or theorem introduced. Event names must satisfy the same restrictions we have on function and variable names in the logic.

The data base is actually an acyclic directed graph with events stored at the nodes. The nodes are named by the event name. In addition to a unique event, each node has several other items of data associated with it, such as, the time at which the event occurred and the names of auxiliary functions or axioms introduced.

The arcs connecting nodes represent “immediate dependency.” The system automatically keeps track of most—but not all (see page 321)—logical dependencies among events. For example, if **SORT** is defined in terms of **INSERT**, then the node for **SORT** is connected to that for **INSERT** by an arc. If **INSERT** is removed from the data base, so is **SORT**.

We discuss the exact form of a node in the Reference Guide on page 319.

7.3. A Summary of the Commands

We here summarize the most commonly used commands. The commands that create events, in alphabetical order, are

- **ADD-AXIOM**: adds a new axiom to the theory and adds the rules generated by it. The **AXIOM** command is an abbreviation for **ADD-AXIOM**.
- **ADD-SHELL**: adds the axioms obtained by instantiating the shell principle and adds the rules generated from those axioms.
- **BOOT-STRAP**: creates the initial data base of axioms and definitions.
- **CONSTRAIN**: introduces a set of new function symbols by constraint, after proving that the constraint is satisfiable.
- **DCL**: declares the arity of a “new” function symbol (making the symbol henceforth “old”) thus permitting its use in formulas.
- **DEFN**: adds a function definition as a new axiom after checking its admissibility and adds the rules generated by the definition.
- **DEFTHEORY**: gives a name to a set of names so that they may be enabled and disabled in unison.
- **FUNCTIONALLY-INSTANTIATE**: derives a theorem by functionally instantiating a previously proved theorem, after proving that the incoming function symbols satisfy the relevant axioms.
- **PROVE-LEMMA**: attempts to prove a conjecture and, if successful, adds to the data base the note that the conjecture is a theorem and adds the rules generated by the theorem. The command **LEMMA** is an abbreviation for a form of **PROVE-LEMMA** in which rules must be explicitly enabled to be used.
- **SET-STATUS**: maps over a theory and sets the enabled/disabled status of each name in it according to a specified function of the type of event that introduced the name. The commands **DISABLE-THEORY** and **ENABLE-THEORY** abbreviate two common forms of **SET-STATUS**.
- **TOGGLE**: enables or disables rules already in the data base. **TOGGLE** commands are also invoked by the **DISABLE** and **ENABLE** commands.
- **TOGGLE-DEFINED-FUNCTIONS**: enables or disables the executable counterparts of defined functions, thus permitting or inhibiting the computation of explicit values on explicit values.

Each event has a name, which is a Lisp symbol. The name of the event associated with **BOOT-STRAP** is **GROUND-ZERO**. The name of a **DEFN** or **DCL** event is the name of the function introduced. The name of the **PROVE-LEMMA** or **ADD-AXIOM** event is the name of the formula, as supplied by the user. The name of an **ADD-SHELL** event is the user-supplied name of the constructor function. Both types of **TOGGLEs** include a user-supplied name among the arguments.

The data base can be queried with the command

- **DATA-BASE**: returns the requested item associated with a given node in the data base.

The command

- **CH**: prints out the most recently executed events and so is useful in determining what rules and concepts have been introduced.

The following commands are of great help when trying to construct a useful data base of rewrite rules. The first command is frequently used when **PROVE-LEMMA** has failed to apply a rule that you expected would be applied; the command can help you determine why the rule was not used.

- **BREAK-LEMMA**: installs a monitor in the rewriter so that when the theorem prover considers the application of a given rule an interactive break occurs. From within this break you can watch the system's attempt to apply the rule.
- **ACCUMULATED-PERSISTENCE**: prints out a list of the most "expensive" rules in the data base, measured in terms of the number of rules tried during the last proof attempt.

The following command provides an execution environment in which terms in the logic can be run on concrete data ("explicit values"):

- **R-LOOP**: reads variable-free terms from the user's terminal and evaluates them in the logic.

The following commands remove nodes from the data base, removing all nodes which depend upon the given one. These commands take event names as arguments:

- **UNDO-NAME**: removes the named event and all of its dependents.
- **UNDO-BACK-THROUGH**: remove the named event and all events executed since the named event.
- **UBT**: a short version of **UNDO-BACK-THROUGH**.

Note. Because dependency tracking is not complete, we do not guarantee that **UNDO-NAME** actually produces a consistent state—it is possible that **UNDO-NAME** fails to find some results that logically depend upon those deleted. But **UNDO-NAME** is a quick and dirty way to experiment with alternative function definitions and theorems. We explain the situation in more detail on page 321.

The following commands process sequences of events, performing each in turn:

- **PROVE-FILE**
- **PROVEALL**
- **DO-EVENTS**

The data base is saved and restored by

- **MAKE-LIB**: writes the current data base to the file system.
- **NOTE-LIB**: loads a data base from the file system.

7.4. Errors

The theorem prover has extensive error checking facilities. If we say a term is expected in the third argument of a command, we mean we cause an error if a term is not supplied. If we say a name has to be “new,” we mean we cause an error if it is already known in the data base. Furthermore, all such errors are checked for before the system makes any changes to the data base. Thus, an error “in the middle” of a command like **ADD-SHELL** or **PROVE-LEMMA**, will not leave the data base in an inconsistent state.

See page 334 in the Reference Guide for a more complete description of error handling.

7.5. Aborting a Command

You will frequently wish to abort a proof attempt because you see that the theorem prover is going astray. We provide no special facility for aborting the execution of a command. However, this feature is provided in most host Lisp systems by typing some particular “control” character. We discuss the details on page 295 in the Reference Guide.

7.6. Syntax and the User Interface

The system is written in Common Lisp. The command interpreter for the system is actually just the top-level Common Lisp “read-eval-print” loop in which Common Lisp s-expressions are read and evaluated and the results are printed. The system’s “commands” are actually just Common Lisp programs. To use the system, you must log onto a computer, enter Common Lisp, load the appropriate files to define the theorem prover primitives, and then call Common Lisp programs to axiomatize concepts, evaluate functions in the logic, submit conjectures to the theorem prover, etc. See the Installation Guide, chapter 14.

This arrangement has its pros and cons. On the pro side, the system’s command interface immediately inherits the sophisticated interactive environment usually available in Lisp (e.g., command editing, histories, read-time abbreviations, output logs, or “dribble files”). In addition, you can mix Common Lisp and theorem prover commands to create new commands, process theorem prover output, etc. On the con side, you must be sufficiently familiar with Common Lisp to type, correct, and execute top-level Lisp forms. In addition, incorrect type-in can throw you into states (such as the debugger or an editor) that may be unfamiliar. Finally, and perhaps most seriously, if you are “malicious” you can “cheat” by using Common Lisp to modify the theorem prover’s programs or state and thus produce the appearance that the theorem prover certifies a result when in fact it does not.

Nevertheless, the theorem prover’s command interpreter *is* Common Lisp and henceforth we assume you are sufficiently familiar with Lisp to operate the theorem prover.

One subtle aspect of this decision is its effect on the syntax of terms in the logic. When you type a command, such as

```
(PROVE-LEMMA PROPERP-REVERSE (REWRITE)
 (PROPERP (REVERSE X)))
```

what actually happens is that the characters you type are read by the Lisp reader and converted into a Lisp s-expression. That s-expression is then evaluated. Thus, our program **PROVE-LEMMA** is actually applied to the s-expressions corresponding to what you typed, not to the character strings themselves. In particular what **PROVE-LEMMA** gets as the formula to be proved is not the character string “(PROPERP (REVERSE X))” but a Lisp list of length 2 whose car is the Lisp symbol **PROPERP**, etc.

It is actually the Lisp reader that implements the syntax we have described for terms. For example, it is the reader that provides the single-gritch **QUOTE** convention, dot notation, flexible use of white space, and the **;** comment convention. When read by the Lisp reader,

```
(APPEND '((A . 0) (B . 1)) ;initial segment
      ALIST)
```

produces exactly the same s-expression as

```
(APPEND (QUOTE ((A . 0) (B . 1))) ALIST).
```

We have carefully defined the syntax of terms so that if you type formulas in the indicated syntax they are read properly by Lisp. See, however, the discussion of **BACKQUOTE-SETTING** in the Reference Guide, page 301.

One other aspect to the syntax is worth noting. Since all of our commands that take “terms” as arguments actually get Lisp s-expressions and the commands themselves are just Lisp programs, commands need not be *typed*. You may wish to write your own interface that simply applies our programs to the appropriate s-expressions. In such an application, the s-expressions can be produced by any Lisp procedure at all, not merely the reader.

7.7. Confusing Lisp and the Logic

A common mistake made by new users is to confuse Lisp and the logic. This confusion commonly takes on two different forms: (a) the meaning of terms in the logic is the same as their meaning in Lisp and (b) the interface to the theorem prover is the logic—i.e., your commands are “evaluated” as applications of logical functions not Lisp s-expressions.

The first misconception is contradicted by many simple examples. In the logic, **(CAR 0)** is **0**, while in Lisp it (usually) causes an error. In the logic, **(ADD1 T)** is **1**, while in Lisp it (usually) causes an error. In the logic, **(IF NIL 1 2)** is **1** because **IF** tests against **F** and **NIL** is not **F**, while in Lisp it is **2** because **IF** tests against **NIL** in Lisp. Finally, in the logic, the equation **(RUSSELL X) = (NOT (RUSSELL X))** is inadmissible as a function definition, while in Lisp that “recipe” for computing **RUSSELL** can be stored as the “definition” of **RUSSELL**. Of course, in Lisp, the attempt to compute **(RUSSELL 3)** “runs forever.”

To illustrate the second misconception, consider

```
(DEFN SECOND (X) (CADR X))
```

First observe that our commands really mix both Lisp and the logic. The word **DEFN** is the name of a Lisp procedure. Its arguments are metalogical objects, namely a symbol, a list of symbols, and a term.

The command interpreter for our theorem proving system is Lisp, not the logic. The effect of the Lisp **DEFN** procedure is to define a function in the logic. It does not define a “function”,¹³ or program in Lisp.

There is no *a priori* relation between the logical function named **fn** and the Lisp program, if any, named **fn**. A function symbol can be used in a term, e.g., to be evaluated by **R-LOOP** or proved by the theorem prover, only if the symbol has been introduced into the logic with **ADD-SHELL**, **BOOT-STRAP**, **CONSTRAIN**, **DCL** or **DEFN**. The Lisp definition, if any, of a symbol is irrelevant to the logic. Symmetrically, a symbol can be applied in Lisp only if it has been defined in Lisp with **DEFUN** or one of the myriad other ways to define function objects in Lisp. The logical definition, if any, of a symbol is irrelevant to Lisp.

¹³In common usage, Lisp programs are frequently called “functions.” We avoid use of the word “function” in that sense, preferring instead “routine,” “procedure,” “command,” or simply “program.”

8 A Sample Session with the Theorem Prover

In this chapter we illustrate several aspects of the system: getting it started, “running” functions in the logic, defining new functions, proving several simple theorems to program the data base, and saving the data base to a file for future use.

Getting the system started is an installation-dependent operation. On some machines one merely types

```
nqthm
```

to the operating system. On others, one might enter for example the locally available version of Common Lisp by typing

```
kcl
```

to the operating system and then load some files that contain the theorem prover code. In Chapter 14 we describe how to obtain the sources for the theorem prover and install them at a new site. For the rest of this chapter we assume the theorem prover has been installed and “started up.”

The theorem prover is nothing but a collection of Lisp programs. The user-interface to the theorem prover is just the read-eval-print loop of Lisp. To use the theorem prover, one must type Lisp expressions that invoke our various commands.

The first thing one must do is load an initial data base. This may be done either by executing the **BOOT-STRAP** command or the **NOTE-LIB** command.

Below we show the invocation of the **BOOT-STRAP** command and the response by the system:

```
(BOOT-STRAP NQTHM)
[ 8.5 11.8 0.0 ]
GROUND-ZERO
```

BOOT-STRAP initializes the system to the Ground Zero logic. It then prints out the “event time,” [8.5 11.8 0.0], a triple of numbers indicating the preprocessing time, cpu time, and output time, in seconds, to carry out this command.¹⁴

The alternative way to initialize the data base is to execute a **NOTE-LIB** command, such as

```
(NOTE-LIB "doe:>nqthm>work")
```

This initializes the system’s data base to that saved in the “library” files **"doe:>nqthm>work.lib"** and **"doe:>nqthm>work.lisp"**.¹⁵ We show how to save a data base later in this section.

To run functions in the logic the command **R-LOOP** is provided. **R-LOOP** enters a “read-eval-print” loop in which the notion of “eval” used is “reduction.” Reduction is the systematic application of the axioms and definitions to reduce a variable-free term to an equivalent explicit value using call-by-value order of evaluation. Every function symbol involved in the process must be defined, introduced by the shell principle, or axiomatized in the Ground Zero logic; applications of constrained and declared function symbols cannot be reduced.

Here is an invocation of **R-LOOP** and the subsequent interaction. The *s below are printed by **R-LOOP** to prompt the user for the next term to be evaluated. Following the * is a term typed by the user. On the next line is the explicit value term to which the input term is equal, or the message that the term was not reducible.

```
(R-LOOP)
Trace Mode: Off   Abbreviated Output Mode: On
Type ? for help.

*(EQUAL T F)
F
```

¹⁴The times shown above are for the Gnu Common Lisp version running on an 8 megabyte Sun 3/60.

¹⁵The format of the file names shown here is that used in the Symbolics file system. The file name formats used by our system are those of its host operating system.

```

*(IF T X Y)
  (NOT REDUCIBLE)
*(IF T 23 45)
  23
*(ADD1 23)
  24
*(MINUS 3)
  -3
*(CONS 1 2)
  '(1 . 2)
*(CAR '(1 . 2))
  1
*(LISTP '(1 . 2))
  T
*(PACK '(65 66 67 . 0))
  'ABC
*(LITATOM NIL)
  T
*(UNPACK NIL)
  '(78 73 76 . 0)
*(APPEND NIL '(E F G))
  '(E F G)
*(APPEND '(A B C D) '(E F G))
  '(A B C D E F G)
*(APPEND '(A B C D . 7) '(E F G))
  '(A B C D E F G)
*OK
Exiting R-LOOP.
NIL

```

The term **OK**, which is a variable symbol, is treated specially by **R-LOOP** as the signal that the user wishes to exit **R-LOOP** and return to the Lisp command interpreter. The value of **R-LOOP** is **NIL**.

The **DEFN** command submits to the system a new proposed definition. For example, the following command defines the function **REVERSE**:

```

(DEFN REVERSE (X)
  (IF (NLISTP X)
    NIL
    (APPEND (REVERSE (CDR X)) (LIST (CAR X))))))

```

The command is equivalent to the axiomatic act

Definition.

```
(REVERSE X)
=
(IF (NLISTP X)
  NIL
  (APPEND (REVERSE (CDR X)) (LIST (CAR X)))).
```

The system's response to this command is

```
Linear arithmetic, the lemmas CDR-LESSEQP and
CDR-LESSP, and the definition of NLISTP inform us
that the measure (COUNT X) decreases according to
the well-founded relation LESSP in each recursive
call. Hence, REVERSE is accepted under the
principle of definition. Observe that:
```

```
(OR (LITATOM (REVERSE X))
  (LISTP (REVERSE X)))
is a theorem.
```

```
[ 0.2 0.4 0.5 ]
REVERSE
```

The paragraph printed concerns two topics. First, it explains why **REVERSE** is admissible. Second, it notes a theorem about the type of **REVERSE**. The theorem states that **REVERSE** returns either a **LITATOM** or a **LISTP**.

Having defined **REVERSE**, we can test it:

```
(R-LOOP)
*(REVERSE NIL)
NIL
*(SETQ X '(A B C D E F G)) ;SETQ is an R-LOOP hack for
  '(A B C D E F G)          ;abbreviating constants
*(REVERSE X)
  '(G F E D C B A)
*(REVERSE (REVERSE X))
  '(A B C D E F G)
*(EQUAL (REVERSE (REVERSE X)) X)
T
*OK
```

We now try to prove the conjecture that **(REVERSE (REVERSE X))** is **X**. The command below submits this conjecture to the theorem prover with the advice that it should be named **REVERSE-REVERSE** and should be built into the data base as a **REWRITE** rule, if it is proved.

```
(PROVE-LEMMA REVERSE-REVERSE (REWRITE)
  (EQUAL (REVERSE (REVERSE X)) X))
```

Give the conjecture the name *1.

We will try to prove it by induction. There is only one plausible induction. We will induct according to the following scheme:

```
(AND
  (IMPLIES (NLISTP X) (p X))
  (IMPLIES (AND (NOT (NLISTP X))
                (p (CDR X)))
            (p X))).
```

Linear arithmetic, the lemmas CDR-LESSEQP and CDR-LESSP, and the definition of NLISTP establish that the measure (COUNT X) decreases according to the well-founded relation LESSP in each induction step of the scheme. The above induction scheme leads to the following two new goals:

```
Case 2. (IMPLIES (NLISTP X)
  (EQUAL (REVERSE (REVERSE X))
        X)),
```

which simplifies, unfolding the definitions of NLISTP and REVERSE, to the new conjecture:

```
(IMPLIES (NOT (LISTP X))
  (EQUAL NIL X)),
```

which has two irrelevant terms in it. By eliminating these terms we get the formula:

F,

which means the proof attempt has

***** F A I L E D *****

```
[ 0.0 0.1 1.5 ]
NIL
```

The reader should be sufficiently familiar with the logic to follow the proof attempt, at least until the system eliminates irrelevant terms. The elimination of irrelevant atomic formulas is just a form of generalization: to prove (**IMPLIES** (**AND** **p q r**) **s**) it is sufficient to prove (**IMPLIES** **r s**), a strategy one might adopt if **p** and **q** seem irrelevant. In the case above, everything seemed irrelevant and the theorem prover adopted the goal **F**—which, if proved, is indeed sufficient to prove the earlier subgoal.

When the system gives up, as above, its output should not be interpreted to mean “the input formula is not a theorem” but merely “the input formula was not proved.” In this case, **PROVE-LEMMA** returns **NIL** and no changes are made to the system’s data base.

Having said that failure does not imply that the input conjecture is faulty, it must be said that the input conjecture for failed proofs is often faulty. The failed proof attempt can sometimes guide the user in the construction of a counterexample. Inspection of the proof attempt above, namely the subgoal preceding **F**

```
(IMPLIES (NOT (LISTP X))
          (EQUAL NIL X))
```

leads to the observation that the input formula is not true if **X** is a non-**LISTP** other than **NIL**. For example, if **X** is **7** then both (**REVERSE** **X**) and (**REVERSE** (**REVERSE** **X**)) are **NIL**, not **7**.

Returning to **R-LOOP**

```
(R-LOOP)
*(REVERSE 7)
NIL
*(SETQ Y (APPEND X 7))
'(A B C D E F G . 7)
*(REVERSE Y)
'(G F E D C B A)
*(EQUAL (REVERSE (REVERSE Y)) Y)
F
*OK
```

The desired property of **REVERSE** only holds if the list being reversed ends in a **NIL**. We therefore next define the predicate that checks whether a list ends in a **NIL**. We call such lists “proper.”

```
(DEFN PROPERP (X)
  (IF (NLISTP X)
      (EQUAL X NIL)
      (PROPERP (CDR X))))
```

Linear arithmetic, the lemmas CDR-LESSEQP and CDR-LESSP, and the definition of NLISTP establish that the measure (COUNT X) decreases according to the well-founded relation LESSP in each recursive call. Hence, PROPERP is accepted under the principle of definition. Observe that:

```
(OR (FALSEP (PROPERP X))
    (TRUEP (PROPERP X)))
```

is a theorem.

```
[ 0.1 0.4 0.5 ]
PROPERP
```

Having defined this concept, we can test it:

```
(R-LOOP)

*(PROPERP NIL)
T

*(PROPERP '(A B C))
T

*(PROPERP '(A B C . ATM))
F

*(PROPERP X)
T

*(PROPERP Y)
F

*OK
```

The desired theorem about **REVERSE** is now proved without assistance. The system does an induction on **X**, generalizes the induction step after using and throwing away the induction hypothesis to produce a conjecture about **REVERSE** and **APPEND**, and then proves this subsidiary conjecture by induction.

```
(PROVE-LEMMA REVERSE-REVERSE (REWRITE)
  (IMPLIES (PROPERP X)
    (EQUAL (REVERSE (REVERSE X))
      X)))
```


Give the conjecture the name *1.

We will try to prove it by induction. Two inductions are suggested by terms in the conjecture. However, they merge into one likely candidate induction. We will induct according to the following scheme:

```
(AND
  (IMPLIES (NLISTP X) (p X))
  (IMPLIES (AND (NOT (NLISTP X))
                (p (CDR X)))
            (p X))).
```

Linear arithmetic, the lemmas CDR-LESSEQP and CDR-LESSP, and the definition of NLISTP inform us that the measure (COUNT X) decreases according to the well-founded relation LESSP in each induction step of the scheme. The above induction scheme leads to the following three new goals:

```
Case 3. (IMPLIES (AND (NLISTP X) (PROPERP X))
  (EQUAL (REVERSE (REVERSE X))
    X)),
```

which we simplify, expanding NLISTP, PROPERP, REVERSE, and EQUAL, to:

T.

```
Case 2. (IMPLIES (AND (NOT (NLISTP X))
  (NOT (PROPERP (CDR X)))
  (PROPERP X))
  (EQUAL (REVERSE (REVERSE X))
    X)).
```

This simplifies, expanding NLISTP and PROPERP, to:

T.

```

Case 1. (IMPLIES
  (AND (NOT (NLISTP X))
    (EQUAL
      (REVERSE (REVERSE (CDR X)))
      (CDR X))
    (PROPERP X))
  (EQUAL (REVERSE (REVERSE X)) X)).

```

This simplifies, opening up the definitions of NLISTP, PROPERP, and REVERSE, to:

```

(IMPLIES
  (AND (LISTP X)
    (EQUAL (REVERSE (REVERSE (CDR X)))
      (CDR X))
    (PROPERP (CDR X)))
  (EQUAL
    (REVERSE (APPEND (REVERSE (CDR X))
      (LIST (CAR X))))
    X)).

```

Applying the lemma CAR-CDR-ELIM, we now replace X by (CONS V Z) to eliminate (CDR X) and (CAR X). We must thus prove:

```

(IMPLIES
  (AND (EQUAL (REVERSE (REVERSE Z)) Z)
    (PROPERP Z))
  (EQUAL
    (REVERSE (APPEND (REVERSE Z) (LIST V)))
    (CONS V Z))).

```

We use the above equality hypothesis by cross-fertilizing (REVERSE (REVERSE Z)) for Z and throwing away the equality. We must thus prove:

```

(IMPLIES
  (PROPERP Z)
  (EQUAL
    (REVERSE (APPEND (REVERSE Z) (LIST V)))
    (CONS V (REVERSE (REVERSE Z))))).

```

We will try to prove the above conjecture by generalizing it, replacing (REVERSE Z) by Y. This generates:

```
(IMPLIES
  (PROPERP Z)
  (EQUAL (REVERSE (APPEND Y (LIST V)))
    (CONS V (REVERSE Y)))).
```

Eliminate the irrelevant term. This produces:

```
(EQUAL (REVERSE (APPEND Y (LIST V)))
  (CONS V (REVERSE Y))),
```

which we will finally name *1.1.

Let us appeal to the induction principle. The recursive terms in the conjecture suggest two inductions. However, they merge into one likely candidate induction. We will induct according to the following scheme:

```
(AND (IMPLIES (AND (LISTP Y) (p (CDR Y) V))
  (p Y V))
  (IMPLIES (NOT (LISTP Y)) (p Y V))).
```

Linear arithmetic and the lemma CDR-LESSP

establish that the measure (COUNT Y) decreases according to the well-founded relation LESSP in each induction step of the scheme. The above induction scheme generates two new conjectures:

```
Case 2. (IMPLIES
  (AND
    (LISTP Y)
    (EQUAL
      (REVERSE (APPEND (CDR Y) (LIST V)))
      (CONS V (REVERSE (CDR Y)))))
  (EQUAL (REVERSE (APPEND Y (LIST V)))
    (CONS V (REVERSE Y)))).
```

which we simplify, rewriting with CAR-CONS and CDR-CONS, and unfolding the definitions of APPEND and REVERSE, to:

T.

```
Case 1. (IMPLIES
  (NOT (LISTP Y))
  (EQUAL (REVERSE (APPEND Y (LIST V)))
    (CONS V (REVERSE Y)))).
```

This simplifies, applying CAR-CONS and CDR-CONS, and expanding the functions APPEND, LISTP, and REVERSE, to:

T.

That finishes the proof of *1.1, which finishes the proof of *1. Q.E.D.

```
[ 0.4 5.6 6.0 ]
REVERSE-REVERSE
```

To review the most recent events we can type

```
(CH 2)
```

The output produced is

```
0  (PROVE-LEMMA REVERSE-REVERSE ...)
1  (DEFN PROPERP ...)
2  (DEFN REVERSE ...)
NIL
```

The events are enumerated backwards. The numbers assigned may be given to **UBT** to “undo-back-through” the indicated position, rolling the system’s state back to that point. Event names may also be used. The final **NIL** is the value delivered by the **CH** program.

We might now try to prove a corollary of **REVERSE-REVERSE**:

```
(PROVE-LEMMA TRIPLE-REVERSE NIL
 (EQUAL (REVERSE (REVERSE (REVERSE A)))
 (REVERSE A)))
```

The first part of the attempt is shown below; however, we interrupt it in the middle of the description of an induction.

Call the conjecture *1.

Perhaps we can prove it by induction. There are two plausible inductions. We [ABORT]

We did not intend for this conjecture to be proved by induction. At first sight, **REVERSE-REVERSE** suffices to prove **TRIPLE-REVERSE** immediately. However, **REVERSE-REVERSE** says that **(REVERSE (REVERSE X))** is **X**, if **(PROPERP X)**. In the intended application, **X** is **(REVERSE A)**. The system tries to relieve the hypothesis that **(PROPERP (REVERSE A))** by recursively rewriting it. However, the fact that **REVERSE** always returns a proper list requires induction to prove. Put another way, the system cannot apply **REVERSE-REVERSE** because the simplifier does not “know” that **REVERSE** builds proper lists. We can remedy the situation by “educating” the simplifier, proving the **REWRITE** rule that **(PROPERP (REVERSE X))**.

This discussion, however, has glossed over a very important point: how, in more complicated settings, do you determine why a given rule was not used? When the “missing rule” is more complicated than **REVERSE-REVERSE** and hundreds of other rules are around to “mess up” the target term and hypotheses, the situation demands some kind of mechanical help. We illustrate that help here by using the **BREAK-LEMMA** facility to investigate the failure of the **TRIPLE-REVERSE** proof attempt.

First, we install a break on the rule in question by typing

```
(BREAK-LEMMA 'REVERSE-REVERSE)
```

Then we invoke the theorem prover again on **TRIPLE-REVERSE**.

```
(PROVE-LEMMA TRIPLE-REVERSE NIL
 (EQUAL (REVERSE (REVERSE (REVERSE A)))
 (REVERSE A)))
```

This time, when **REVERSE-REVERSE** is tried, we get an interactive break:

```
(Entering break on replacement rule
REVERSE-REVERSE
```

The “break package” prints a colon as the prompt character and waits for user input. We type a “?” which causes a brief summary of the available commands to be printed:

```
: ?
You are in the BREAK-REWRITE command interpreter.
The commands specific to this break are:

      cmd          effect
OK      Attempt to relieve the hypotheses
         and then break.

GO      Proceed without further interaction.
NAME    Print name of the broken rule.
HYPS    List the hypotheses of the broken
         rule.

CONCL   Print the conclusion of the broken
         rule.
SUBST   Print the substitution being applied
         to the broken rule.
TARGET  Print the term to which the rule is
         being applied.
??      General purpose commands.
```

We then type the commands to display the conclusion of the rule, the hypotheses of the rule, and the target term to which the rule is being applied:

```
: concl
(EQUAL (REVERSE (REVERSE X)) X)
: hyps
1. (PROPERP X)
: target
(REVERSE (REVERSE A))
```

Observe that the rule is being applied to the innermost nest of **REVERSE**s. To apply the rule we will have to establish that **A** is **PROPERP**. This is not possible since we know nothing about **A**. Nevertheless, we type the command **ok** to observe the attempt to relieve the hypothesis. Upon seeing the explanation of

the failure, we type **go** which causes the system to complete the attempt to apply this rule and continue to simplify the goal formula.

```

: ok
Attempting to relieve the hypotheses of
  REVERSE-REVERSE...
Failed to establish the first hypothesis of
  REVERSE-REVERSE:
  (PROPERP X)
under the substitution
X <- A
because the hypothesis rewrote to:
  (PROPERP A)
: go
Application of REVERSE-REVERSE failed.
Exiting break on replacement rule
REVERSE-REVERSE.)

```

The rule is still being monitored and another break occurs. This time when we print the **target** term we see that it is the outer **REVERSE** nest, the one to which we expected to apply the rule. The **ok** command causes the system to try to relieve the hypothesis, and we see that it simplifies to **(PROPERP (REVERSE A))**, not **T**. Thus, we know why **REVERSE-REVERSE** was not applied.

```

(Entering break on replacement rule
  REVERSE-REVERSE
: target
(REVERSE (REVERSE (REVERSE A)))
: ok
Attempting to relieve the hypotheses of
  REVERSE-REVERSE...
Failed to establish the first hypothesis of
  REVERSE-REVERSE:
  (PROPERP X)
under the substitution
X <- (REVERSE A)
because the hypothesis rewrote to:
  (PROPERP (REVERSE A))
: go
Application of REVERSE-REVERSE failed.
Exiting break on replacement rule
REVERSE-REVERSE.)

```

Upon exiting the proof attempt continues and we abort it, as before:

Call the conjecture *1.

Perhaps we can prove it by induction.
There [ABORT]

Having discovered why **REVERSE-REVERSE** is not applied, we remove the break on it:

(UNBREAK-LEMMA 'REVERSE-REVERSE)

and proceed to "educate" the simplifier by proving the general theorem that (REVERSE X) is PROPERP for all X.

(PROVE-LEMMA REVERSE-IS-PROPERP (REWRITE)
(PROPERP (REVERSE X)))

Give the conjecture the name *1.

Let us appeal to the induction principle.
There is only one suggested induction. We will
induct according to the following scheme:

(AND
 (IMPLIES (NLISTP X) (p X))
 (IMPLIES (AND (NOT (NLISTP X))
 (p (CDR X)))
 (p X))).

Linear arithmetic, the lemmas CDR-LESSEQP and CDR-LESSP, and the definition of NLISTP inform us that the measure (COUNT X) decreases according to the well-founded relation LESSP in each induction step of the scheme. The above induction scheme generates two new conjectures:

Case 2. (IMPLIES (NLISTP X)
 (PROPERP (REVERSE X))),

which simplifies, opening up the definitions of NLISTP, REVERSE, and PROPERP, to:

T.


```

Case 1. (IMPLIES
        (AND (NOT (NLISTP X))
              (PROPERP (REVERSE (CDR X)))))
        (PROPERP (REVERSE X))).

```

This simplifies, expanding the functions NLISTP and REVERSE, to:

```

(IMPLIES
 (AND (LISTP X)
       (PROPERP (REVERSE (CDR X)))))
 (PROPERP (APPEND (REVERSE (CDR X))
                  (LIST (CAR X))))).

```

Applying the lemma CAR-CDR-ELIM, we now replace X by (CONS V Z) to eliminate (CDR X) and (CAR X). We thus obtain:

```

(IMPLIES (PROPERP (REVERSE Z))
         (PROPERP (APPEND (REVERSE Z)
                          (LIST V)))),

```

which we generalize by replacing (REVERSE Z) by Y. We thus obtain:

```

(IMPLIES (PROPERP Y)
         (PROPERP (APPEND Y (LIST V)))),

```

which we will name *1.1.

We will appeal to induction. There are two plausible inductions. However, they merge into one likely candidate induction. We will induct according to the following scheme:

```

(AND
 (IMPLIES (NLISTP Y) (p Y V))
 (IMPLIES
  (AND (NOT (NLISTP Y))
        (p (CDR Y) V))
  (p Y V))).

```

Linear arithmetic, the lemmas CDR-LESSEQP and CDR-LESSP, and the definition of NLISTP inform us that the measure (COUNT Y) decreases according to the well-founded relation LESSP in each induction step of the scheme. The above induction scheme leads to the following three new formulas:

*(The proofs of the three cases
have been deleted for brevity.)*

That finishes the proof of *1.1, which,
consequently, finishes the proof of *1. Q.E.D.

```
[ 0.0 0.8 4.2 ]
REVERSE-IS-PROPERP
```

At this point we reconsider the proof of **REVERSE-IS-PROPERP** and decide that the proof is prettier if the relationship between **APPEND** and **PROPERP** is proved first as a lemma. Therefore, the just proved **REVERSE-IS-PROPERP** is “undone”—removed from the data base:

```
(UNDO-NAME 'REVERSE-IS-PROPERP)
((PROVE-LEMMA REVERSE-IS-PROPERP (REWRITE)
  (PROPERP (REVERSE X))))
```

The **UNDO-NAME** command returns a list of the commands that have been removed from the data base.

Next we formulate the necessary relationship between **PROPERP** and **APPEND**.

```
(PROVE-LEMMA APPEND-IS-PROPERP (REWRITE)
  (IMPLIES (PROPERP B) (PROPERP (APPEND A B))))
```

The proof is by induction on **A** but is omitted here. Actually, the future behavior of the system would be more efficient if we had instead proved

```
(EQUAL (PROPERP (APPEND A B))
  (PROPERP B))
```

since, when used as a **REWRITE** rule, the equality eliminates **APPEND** without backwards chaining.

Now, revisiting **REVERSE-IS-PROPERP**, we see the proof is much simpler:

```
(PROVE-LEMMA REVERSE-IS-PROPERP (REWRITE)
  (PROPERP (REVERSE X)))
```

Give the conjecture the name *1.

Let us appeal to the induction principle.
There is only one suggested induction. We will
induct according to the following scheme:

```
(AND
  (IMPLIES (NLISTP X) (p X))
  (IMPLIES (AND (NOT (NLISTP X))
                (p (CDR X)))
            (p X))).
```

Linear arithmetic, the lemmas CDR-LESSEQP and
CDR-LESSP, and the definition of NLISTP inform us
that the measure (COUNT X) decreases according to
the well-founded relation LESSP in each induction
step of the scheme. The above induction scheme
generates two new conjectures:

```
Case 2. (IMPLIES (NLISTP X)
                 (PROPERP (REVERSE X))),
```

which simplifies, opening up the definitions of
NLISTP, REVERSE, and PROPERP, to:

T.

```
Case 1. (IMPLIES
  (AND (NOT (NLISTP X))
        (PROPERP (REVERSE (CDR X)))))
  (PROPERP (REVERSE X))).
```

This simplifies, applying CDR-CONS and
APPEND-IS-PROPERP, and opening up the
definitions of NLISTP, REVERSE, and PROPERP,
to:

T.

That finishes the proof of *1. Q.E.D.

```
[ 0.0 0.1 1.4 ]
REVERSE-IS-PROPERP
```

Our identification of **APPEND-IS-PROPERP** as a useful **REWRITE** rule is typical of the way the informed user can cause the system to produce simple proofs.

Finally, having laid the proper groundwork, we return to the **TRIPLE-REVERSE** conjecture:

```
(PROVE-LEMMA TRIPLE-REVERSE (REWRITE)
  (EQUAL (REVERSE (REVERSE (REVERSE A)))
    (REVERSE A)))
```

This simplifies, appealing to the lemmas **REVERSE-IS-PROPERP** and **REVERSE-REVERSE**, to:

T.

Q.E.D.

```
[ 0.1 0.1 0.1 ]
TRIPLE-REVERSE
```

To save the data base we can execute

```
(MAKE-LIB "demo")
```

which will write two files, one called **"demo.lib"** and the other called **"demo.lisp"**.¹⁶ Having saved the data base, we could log out, reset the machine, boot, or otherwise destroy the current machine configuration and recover an equivalent state, as far as the theorem prover is able to determine, by starting up a fresh copy of the theorem prover and invoking

```
(NOTE-LIB "demo").
```

¹⁶Actually, the unique, full names of the files created depends upon the setting of several variables and the defaults used by the host file system. See the discussion of File Names in the Reference Guide, page 339.

9 How to Use the Theorem Prover

This chapter might also be called **How To Use the Theorem Prover to Discover Proofs** or **How to Interact with the Theorem Prover** or **How to Read Theorem Prover Output**.

Recall the example proof of **REVERSE-REVERSE**: the theorem prover did an induction, simplified the induction step, eliminated a destructor, used an equality hypothesis in a heuristic way and threw it away, generalized a common subterm, discarded an irrelevant hypothesis, and did a second induction. This example is misleading in one respect: few users of the theorem prover let the system even try to produce such complicated proofs.

Successful users of the theorem prover tend to arrange things so that most proofs fall into one of two forms:

- induction followed by simplification of each case, or
- simplification alone.

Complicated proofs are broken down into lemmas or cases, and each case is proved in isolation and stored as a **REWRITE** rule. Thus, the theorem prover is kept “on a short leash.” This is an attractive strategy for many reasons. First, the system is more predictable—less is expected of it and so it more often meets your expectations. Second, you’ll spend less time analyzing unexpected subgoals to see if the system has a chance—when it deviates from the expected path, blow it out of the water with the **ABORT** key. Third, you’ll spend less time trying to trick it into producing the entire proof “automatically.” Fourth, once a

subgoal has been enshrined as a rule in the data base you will not have to prove it again—seeing the system wind its way through a complicated proof of an early subgoal is impressive if the entire proof attempt succeeds, but it is maddening if a later subgoal makes the entire proof attempt fail. Finally, when you are all done, the resulting data base is useful for future problems in the same domain.

We call this style of use “cooperative” because the final proof structure is produced jointly by the user and the theorem prover. The key to using the theorem prover cooperatively is to *know when to scrutinize its output* and to know what to do when you want to change the structure of the emerging proof. The crucial “check points” are selection of the induction schema, transition from simplification to the other processes, attempted generalization, and the naming of a subgoal to be proved by induction.

The remainder of this chapter is organized as follows. First we illustrate a cooperative session with the theorem prover by redoing the proof of **REVERSE-REVERSE** in this style. Then we discuss how we use the traditional Lisp interface and text editor to carry out such a session. Finally, we give general guidelines for what to think about at each of the checkpoints.

9.1. REVERSE-REVERSE Revisited—Cooperatively

Suppose we are in the **GROUND-ZERO** state except that **REVERSE** and **PROPERP** have been defined. We wish to prove **REVERSE-REVERSE**:

```
(PROVE-LEMMA REVERSE-REVERSE (REWRITE)
  (IMPLIES (PROPERP X)
    (EQUAL (REVERSE (REVERSE X)) X))).
```

First, we execute the above command. The induction selected by the system, on the **CDR** of **X**, is appropriate. The proof of the induction step, **Case 1**, starts as follows:

```
Case 1. (IMPLIES
  (AND (NOT (NLISTP X))
    (EQUAL
      (REVERSE (REVERSE (CDR X)))
      (CDR X))
    (PROPERP X))
  (EQUAL (REVERSE (REVERSE X)) X)).
```

This simplifies, opening up the definitions of **NLISTP**, **PROPERP**, and **REVERSE**, to:

```
(IMPLIES
  (AND (LISTP X)
    (EQUAL (REVERSE (REVERSE (CDR X)))
      (CDR X))
    (PROPERP (CDR X)))
  (EQUAL (REVERSE
    (APPEND (REVERSE (CDR X))
      (LIST (CAR X))))
    X)).
```

Appealing to the lemma **CAR-CDR-ELIM**, we now ...

Note that the last formula printed was produced by simplification, put into the pool, drawn out by the simplifier, and *did not change*. It thus entered the destructor elimination process. The transition from simplification to any of the later processes is critical. We call this a “check point” and there are, in all, four of them.

At this particular check point you should ask yourself the question “Are there some facts about the combinations of functions in this subgoal that could be used to further simplify the formula?” In the case above, the answer is “yes, **REVERSE** distributes over **APPEND**, sort of.”

In particular,

```
(PROVE-LEMMA REVERSE-APPEND (REWRITE)
  (EQUAL (REVERSE (APPEND A B))
    (APPEND (REVERSE B) (REVERSE A)))).
```

It happens that if this fact is used as a rewrite rule during the simplification above, the subgoal simplifies to **T** and the proof of **REVERSE-REVERSE** is complete. However, it is not necessary that you recognize the sufficiency of **REVERSE-APPEND** as long as you recognize its utility.

Having diagnosed the need for **REVERSE-APPEND**, you should abort the proof attempt of **REVERSE-REVERSE** and focus instead on proving **REVERSE-APPEND**. That is, issue the **PROVE-LEMMA** command for **REVERSE-APPEND**.

The system inducts on **A** by **CDR**, as appropriate. There is a base case and an induction step. The base case transits from simplification (all the way to generalization) on the formula

```
(IMPLIES (NOT (LISTP A))
  (EQUAL (REVERSE B)
    (APPEND (REVERSE B) NIL)))
```


Again, study the formula at the transition. Is there some fact we know that permits this to be simplified further? “Yes, **NIL** is a right-identity for **APPEND**, on **PROPERP** lists, and **REVERSE** produces a **PROPERP** list.” We thus single out two facts:

```
(PROVE-LEMMA APPEND-RIGHT-ID (REWRITE)
  (IMPLIES (PROPERP X)
    (EQUAL (APPEND X NIL) X)))
```

and

```
(PROVE-LEMMA PROPERP-REVERSE (REWRITE)
  (PROPERP (REVERSE X))).
```

Again, abort the proof attempt of **REVERSE-APPEND** and try proving these two goals.

The first is proved without difficulty—that is, the system inducts, and simplification reduces each case to **T**.

Next, submit the command for the second goal, **PROPERP-REVERSE**. An induction on **X** is chosen. The base case simplifies to **T**. The induction step simplifies and transits to destructor elimination on the formula:

```
(IMPLIES (AND (LISTP X)
  (PROPERP (REVERSE (CDR X))))
  (PROPERP
    (APPEND (REVERSE (CDR X))
      (LIST (CAR X))))).
```

Again we ask what we know about the function nests involved. The answer is “**APPEND** constructs a **PROPERP** list if its second argument is **PROPERP**.” Formally, this relation is expressed

```
(IMPLIES (PROPERP Y)
  (PROPERP (APPEND X Y))).
```

However, it is good to write unconditional replacement rules when possible, and a stronger relationship comes to mind: “**APPEND** constructs a **PROPERP** list if and only if its second argument is **PROPERP**.”

Thus, we submit

```
(PROVE-LEMMA PROPERP-APPEND (REWRITE)
  (EQUAL (PROPERP (APPEND X Y))
    (PROPERP Y))).
```

This is proved without difficulty.

We return to the proof of **PROPERP-REVERSE** by executing the **PROVE-LEMMA** command again:

```
(PROVE-LEMMA PROPERP-REVERSE (REWRITE)
  (PROPERP (REVERSE X))).
```

The proof is completed without difficulty now, since **PROPERP-APPEND** simplifies the induction step to **T**.

We thus return to **REVERSE-APPEND**, knowing that the base case will now simplify to **T** since we have that **NIL** is a right-identity for **APPEND** on **PROPERP** lists and **REVERSE** is **PROPERP**. Re-executing the **PROVE-LEMMA** command we see the same induction and the expected proof of the base case. However, the induction step transits from simplification to destructor elimination on

```
(IMPLIES
  (AND (LISTP A)
    (EQUAL (REVERSE (APPEND (CDR A) B))
      (APPEND (REVERSE B)
        (REVERSE (CDR A))))))
  (EQUAL (APPEND (REVERSE (APPEND (CDR A) B))
    (LIST (CAR A)))
    (APPEND (REVERSE B)
      (APPEND (REVERSE (CDR A))
        (LIST (CAR A)))))).
```

Is there anything we know about the function nests here? Facts about nests of **APPENDS** and **REVERSES** are uninteresting since we are trying to prove the fundamental one. Nothing comes to mind and so we let the proof continue. The destructor elimination takes place, the equality hypothesis is used, and the following formula arrives at the generalization process:

```
(EQUAL (APPEND (APPEND (REVERSE B) (REVERSE X))
  (LIST Z))
  (APPEND (REVERSE B)
    (APPEND (REVERSE X) (LIST Z)))).
```

Always read carefully any formula to be generalized and look for relevant **REWRITE** rules for it.

Here an obvious rule suggests itself: “**APPEND** is associative.”

```
(PROVE-LEMMA APPEND-IS-ASSOCIATIVE (REWRITE)
  (EQUAL (APPEND (APPEND A B) C)
    (APPEND A (APPEND B C)))).
```

This is proved without difficulty.

If we now reissue the **PROVE-LEMMA** command for **REVERSE-APPEND** the proof is straightforward.

To conclude, we give the command to prove **REVERSE-REVERSE** and the proof is done without difficulty.

The final list of commands in the order they were successfully processed is

```
(DEFN REVERSE (X)
  (IF (NLISTP X)
      NIL
      (APPEND (REVERSE (CDR X)) (LIST (CAR X)))))

(DEFN PROPERP (X)
  (IF (NLISTP X)
      (EQUAL X NIL)
      (PROPERP (CDR X))))

(PROVE-LEMMA APPEND-RIGHT-ID (REWRITE)
  (IMPLIES (PROPERP X)
            (EQUAL (APPEND X NIL) X)))

(PROVE-LEMMA PROPERP-APPEND (REWRITE)
  (EQUAL (PROPERP (APPEND X Y))
          (PROPERP Y)))

(PROVE-LEMMA PROPERP-REVERSE (REWRITE)
  (PROPERP (REVERSE X)))

(PROVE-LEMMA APPEND-IS-ASSOCIATIVE (REWRITE)
  (EQUAL (APPEND (APPEND A B) C)
          (APPEND A (APPEND B C))))

(PROVE-LEMMA REVERSE-APPEND (REWRITE)
  (EQUAL (REVERSE (APPEND A B))
          (APPEND (REVERSE B) (REVERSE A))))

(PROVE-LEMMA REVERSE-REVERSE (REWRITE)
  (IMPLIES (PROPERP X)
            (EQUAL (REVERSE (REVERSE X)) X)))
```

Below we list the commands submitted for execution and indicate whether the execution terminated with success. We indent to indicate the implicit subgoal structure of the session:

REVERSE	succeeded
PROPERP	succeeded
REVERSE-REVERSE	aborted
REVERSE-APPEND	aborted
APPEND-RIGHT-ID	succeeded
PROPERP-REVERSE	aborted
PROPERP-APPEND	succeeded
PROPERP-REVERSE	succeeded
REVERSE-APPEND	aborted
APPEND-IS-ASSOCIATIVE	succeeded
REVERSE-APPEND	succeeded
REVERSE-REVERSE	succeeded

Each successful proof produced above is of the form “induct and simplify each case” with the exception of the proof for **REVERSE-APPEND** which requires a destructor elimination and heuristic equality substitution to set up an application of the associativity of **APPEND**.¹⁷ Note that when we finish we have a useful library of rules about **APPEND**, **REVERSE**, and **PROPERP**. For example, without further difficulty we can prove the **TRIPLE-REVERSE** theorem.

9.2. Using Lisp and a Text Editor as the Interface

The theorem proving session illustrated above is typical of the current use of the system. Of course, few sessions are so strict in their adherence to a particular methodology. In many sessions we might have let the system “have its head” and thus have let it stumble upon complicated proofs of some of our lemmas simply because we felt it could manage and we had other things to think about.

A question that immediately comes to mind, however, is how the cooperative user manages to remember all the pending theorems and subgoals. Clearly some kind of stacklike bookkeeping mechanism is called for. However, what the user community has evolved so far is not so much a bookkeeping mechanism as just a style of using the text editing environments usually provided by host systems.

¹⁷The elimination of the destructors here is strictly unnecessary: we could have immediately carried out the equality substitution setting up the conclusive application of the associativity of **APPEND**. Often you will see such unnecessary proof steps. The problem is that the system tries its heuristics in a certain order.

In most systems it is possible to type text into an “edit buffer,” evaluate part of the buffer in the Lisp system—possibly while “in” the editor but otherwise by “grabbing” the text and carrying it to the Lisp environment—and shipping the output to both the terminal and another edit buffer (here called the “session log”).

Using these standard features we illustrate our own bookkeeping style by describing how the above session could be carried out. Start with an empty edit buffer. Type into it the **PROVE-LEMMA** command for **REVERSE-REVERSE**. Execute that command. Diagnose the need for **REVERSE-APPEND** and abort the proof attempt. Type the **PROVE-LEMMA** command for **REVERSE-APPEND** into the edit buffer immediately before the command just tried. Execute it. Diagnose the need for **APPEND-RIGHT-ID** and **PROPERP-REVERSE** and abort the proof attempt. Type the **PROVE-LEMMA** commands for **APPEND-RIGHT-ID** and **PROPERP-REVERSE** into the buffer immediately before the command just tried. Execute the first command. It succeeds. Execute the next command in the buffer. Diagnose the need for **PROPERP-APPEND** and abort the proof attempt. Type the **PROVE-LEMMA** command for **PROPERP-APPEND** into the buffer immediately before the command just tried. Execute it. It succeeds. Execute the next command (**PROPERP-REVERSE**). It succeeds. Execute the next command (**REVERSE-APPEND**). Diagnose the need for **APPEND-IS-ASSOCIATIVE** and abort the proof attempt. Type the **PROVE-LEMMA** command for **APPEND-IS-ASSOCIATIVE** into the buffer immediately before the command just tried. Execute it. It succeeds. Execute the next command (**REVERSE-APPEND**). It succeeds. Execute the next command (**REVERSE-REVERSE**). It succeeds. The final state of the edit buffer is that it has, in order, the successful events starting with **APPEND-RIGHT-ID** and ending with **REVERSE-REVERSE**.

We have experimented with other bookkeeping mechanisms to support this subgoal structure. However, nobody uses them; we prefer the text editor approach. The reason is that the evolving buffer of commands soon becomes heavily annotated and less structured than suggested above. For example, explanations and comments are interspersed between commands, alternative command sequences tried and abandoned are recorded for posterity, and temporary notes are tacked up in the buffer:

```
??? I am stuck here. I made a LIB file and I'm going
    home for the night. The next proof needs a lemma
    relating FETCH and LINK. I don't see what it is.
```

Finally, the edit buffer can be written out as a file and read as a coherent description of the current state. Indeed, in the absence of a library file, the system's data base can be rapidly reconstructed (up to proof-based logical

dependencies) from the event file with **SKIM-FILE** (see page 385 in the Reference Guide).

9.3. The Crucial Check Points in a Proof Attempt

There are four crucial “check points” in a proof attempt. You should learn to recognize these points and know what to look for when a proof attempt reaches one of them.

9.3.1. *Induction Selection*

The first check point is when the system chooses an induction argument for a formula. Is the induction argument appropriate for the formula? We cannot pose this question in a more concrete form because the system selects the best induction we know how to manufacture from the terms involved. Usually you will have a particular induction argument in mind—for better or for worse—and all that is required is to determine whether the system used that argument. To force the selection of your induction argument, use the **INDUCT** hint of **PROVE-LEMMA**. A strategy we often adopt is to let the system proceed with its chosen induction and wait until a later check point to think about it. If we are confronted with a goal we don’t see how to prove—“I don’t see why this hypothesis implies that conclusion.”—we go back to the output log and consider the induction more carefully.

9.3.2. *Transition Out of Simplification*

The second check point is when the simplification of a formula is complete and the processing transits to destructor elimination or one of the later processes. This is the check point at which you will spend most of your time. Formulas at this point have an important property: they are stable under simplification. They would not have made the transition to the later process had they been subject to further simplification. Thus a **REWRITE** rule can be designed to match a subterm of such a formula without having to worry that some sub-expression of the target will undergo additional simplification before the rule is tried.¹⁸

¹⁸Unless, of course, the subexpression is rewritten by the new rule itself!

Questions you should ask include the following: Are there additional facts about the function nests in the formula that could be used to further simplify the formula? Typically the appropriate response is to recognize the need for a **REWRITE** rule, abort the current proof, and prove the new rule before restarting. We initially look for “local” simplifications—ones that suggest themselves without detailed consideration of the hypotheses. For example, we consider simplifications of each functional composition ($\mathbf{f} \ (\mathbf{g} \ \dots) \ \dots$) in the formula. If no such “local” simplifications suggest themselves, we usually ask “why is this formula a theorem?” and often see “nonlocal” reasons: two hypotheses are contradictory, a chain of lemmas must be invoked, or additional processing (e.g., destructor elimination or use of an equality hypothesis) is necessary.

It is sometimes the case that the necessary lemma has already been proved but is not being applied by the simplifier. Ask the following questions: Was it proved as a **REWRITE** rule? Is it currently enabled? What is the form of the rule generated? Does it match with a term in the simplified conjecture? If so, which hypothesis of the rule is not being established? Remember: a rule will be applied only if (a) its left-hand side matches a term being rewritten and (b) its hypotheses are established. If you want a rule to be applied, you must arrange for both of those things to happen.

This check point is *the* place to consider whether your rules match the terms in the conjecture. Asking that question earlier or later is often pointless: terms you see in other conjectures are destined to be rewritten and will have changed by the time your new rule is applied.

If the rule matches but is not firing it is almost certainly because some hypothesis cannot be established. We illustrated this in the first proof attempt for **TRIPLE-REVERSE** (page 242). In general, when a hypothesis is not being relieved, the first course of action is to prove a rule that establishes it.

If you cannot arrange for a rule to fire automatically, consider using the **USE** hint to **PROVE-LEMMA**.

The other commonly arising problem is that a definition fails to expand. The most convenient response is to use the **EXPAND** hint in the **PROVE-LEMMA** command. That hint requires that you provide exactly the term to be expanded, and this check point is *the* place to determine it.

In addition to considering the state of your **REWRITE** rules, the transition out of simplification should cause you to consider other issues. It is at this point that we often recognize that an inappropriate induction was done, that the conjecture submitted is not strong enough to go through an inductive proof, or that the conjecture submitted is not a theorem.

If we are unable to see any additional simplifications and do not have reason to doubt the induction or the validity of the conjecture, we usually let the system continue until the next check point.

9.3.3. *Generalization*

The third check point is when a formula is generalized. We almost never permit the system to generalize a formula for us. Many times we automatically abort when we see the words “**which we generalize.**” The question we ask ourselves here is exactly the same as at the transition out of simplification: what additional facts would permit this formula to simplify? But assuming we thought about that question at the last transition, we can broaden the question to “why is this formula a theorem?” That is, we’re not interested here in just further simplifying it but rather in proving it. Typical responses are to identify a more general theorem that subsumes the current conjecture—one that is at the heart of the proof of the main conjecture. Things to look for are contradictory hypotheses or, more generally, generalizations of fragments of the given conjecture. For example, the first and fifth hypotheses, slightly generalized, may imply the conclusion, slightly generalized.

9.3.4. *Pushing a Subgoal for Induction*

The fourth check point is when a formula is given a name (e.g., “***1.4**”) and pushed into the stack of things to be proved by induction. At the very least, ask yourself “Is the subgoal a theorem?” We more often simply abort—assuming we’ve already done one induction—and consider more carefully the **REWRITE** rules necessary to prove the conjecture earlier in simplification.

If you permit the system to proceed, it will attack these named formulas by induction. Therefore, a subsidiary question is “Is the subgoal sufficiently general to be proved by induction?” Often this will force you to invent a formula that is more general than the ones you’ve been thinking about. The formula will be provable by induction and, as a **REWRITE** rule, will simplify the current subgoal to **T**.

10 How the Theorem Prover Works

In this chapter we describe how the theorem prover works, with particular attention to those aspects of its behavior under the control of the user through previously proved theorems. Recall that when theorems are proved by the system the user is responsible for declaring what classes of rules are to be generated from the theorem. The four classes of rules are **REWRITE**, **META**, **ELIM**, and **GENERALIZE**. We note in the discussion below where the various classes of rules are used. In the next chapter we discuss in detail the generation and interpretation of rules.

Recall that the theorem prover is built out of six processes: simplification, destructor elimination, cross-fertilization (heuristic equality use), generalization, elimination of irrelevance, and induction. Each takes a formula as input and returns a set of formulas as output. When one process does not change the input formula, the next process in the list above is tried. We now discuss each of the six processes in turn, emphasizing those that are sensitive to previously proved theorems. However, we first discuss two more general issues, the representation of formulas inside the processes and the use of type information.

10.1. The Representation of Formulas

The formulas manipulated by our processes are kept in “clausal” form. A *clause*, in our sense, is a list of terms ($\text{lit}_1 \text{ lit}_2 \dots \text{lit}_n$) and represents the formula

$$\text{lit}_1 \neq \mathbf{F} \vee \text{lit}_2 \neq \mathbf{F} \vee \dots \vee \text{lit}_n \neq \mathbf{F}$$

Each lit_i is called a *literal*.¹⁹

For example, when the user submits the following conjecture

```
(IMPLIES (AND (PRIME P)
               (NOT (DIVIDES P M)))
          (EQUAL (REMAINDER (EXPT M (SUB1 P)) P)
                 1))
```

it is actually a *term* that is submitted. The term is converted into the following clause:

```
((NOT (PRIME P))
 (DIVIDES P M)
 (EQUAL (REMAINDER (EXPT M (SUB1 P)) P) 1))
```

The simplifier and all other processes actually manipulate this clausal form of the conjecture. The advantage to clausal form is that it encourages symmetric treatment of hypotheses and conclusions.

The conversion to clausal form is usually invisible to the user because the theorem prover prints clauses by first applying the inverse conversion, mapping the clause back into the form $(\text{IMPLIES } (\text{AND } h_1 \dots h_n) \text{ concl})$ before printing it. However, understanding the clausal form underlying the printed representation of formulas helps the user who is designing rules. For example, since the **AND** in the hypothesis of the implication above is not actually in the clausal form, no **REWRITE** rule designed to hit $(\text{AND } h_1 \dots h_n)$ will touch it.

¹⁹Traditionally, a literal is either an atomic formula or the negation of an atomic formula, rather than a term. A clause is traditionally a set, rather than a list, of literals.

10.2. Type Sets

Type sets indicate the shell classes over which the value of an expression ranges and are used extensively throughout the theorem prover.

Without loss of generality, let us assume that all the shells that will ever be added have been added and that the resulting history contains n shell recognizers, r_1, \dots, r_n , the first six of which are **FALSEP**, **TRUEP**, **NUMBERP**, **LITATOM**, **LISTP**, and **NEGATIVEP**. Let us also assume that

$$\begin{aligned} &(\text{OTHERS } x) \\ &= \\ &(\text{AND } (\text{NOT } (r_1 \ x)) \ \dots \ (\text{NOT } (r_n \ x))) \end{aligned}$$

Let Ω be the set $\{r_1 \dots r_n \text{ OTHERS}\}$. Then a *type set* is any subset of Ω . To say that a term t has type set $\{s_1 \dots s_k\}$ means that $(\text{OR } (s_1 \ t) \dots (s_k \ t))$ is a theorem. For example, the term $(\text{IF } (\text{LISTP } x) (\text{CONS } a \ b) \ \text{NIL})$ has type set $\{\text{LISTP } \text{LITATOM}\}$. The term $(\text{IF } (\text{NUMBERP } x) \ T \ x)$ has type set $\Omega - \{\text{NUMBERP}\}$.

The most notable use of type sets in the theorem prover is to encode the assumptions under which the theorem prover is working. As the rewriter explores a term, it maintains a context that encodes the current assumptions as type sets. For example, consider the problem of exploring an expression of the form $(\text{IF } \text{test } \text{left } \text{right})$. While working on **left** it is permitted to assume that **test** is non-**F**, and while working on **right** it is permitted to assume that **test** is **F**. How do we assume **test** is “non-**F**” when working on **left**? Suppose we knew initially that the type set of **test** was ts . Then when working on **left** we could assign **test** the type set $ts - \{\text{FALSEP}\}$. When working on **right** we assign **test** the type set $\{\text{FALSEP}\}$.

Some **tests** permit us finer-grained control of the type sets. Suppose **test** is $(\text{NUMBERP } x)$ and that under the current assumptions the type set of x is ts_x . If ts_x is $\{\text{NUMBERP}\}$, then **test** is **T**; if ts_x does not include **NUMBERP**, then **test** is **F**. Otherwise, when working on **left** we may assume that the type set of x is $\{\text{NUMBERP}\}$, and while working on **right** we may assume that it is $ts_x - \{\text{NUMBERP}\}$. In fact this type of processing is generalized for user-defined predicates about which certain kinds of formulas have been proved. By proving the appropriate “compound recognizer” rules (see page 284) it can be arranged, for example, for the assumption of $(\text{BOOLEANP } x)$ to set the new type set of x to $\{\text{TRUEP } \text{FALSEP}\}$ on the left branch and to everything but $\{\text{TRUEP } \text{FALSEP}\}$ on the right.

Because type sets are a general purpose way of encoding assumptions, they are used in virtually every process of the theorem prover. Such fundamental questions as “Is this term non-**F**?” send the theorem prover off into type set computation.

When functions are defined by the user, the system computes a “type prescription rule” (see page 282) that describes how a type set can be computed for any application of the function.²⁰ The definitional mechanism prints out a formula that describes the type prescription rule generated. The user can override the generated rule by proving certain forms of **REWRITE** rules that establish smaller type sets. For example, one might prove **(NUMBERP (GCD X Y))** as a **REWRITE** rule, with the effect that the type set for **GCD** expressions would henceforth be computed to be **{NUMBERP}**.

10.3. Simplification

The simplifier is the most complex process in the theorem prover. It is the only process that can return the empty set of formulas—i.e., it is the only process that can establish a goal instead of merely transforming it into subgoals. The simplifier combines many different proof techniques: decision procedures for propositional calculus and equality, a procedure for deciding many of the truths of linear arithmetic over the natural numbers, a term rewriting system, and the application of metafunctions.

Recall that the simplifier takes as input a formula and produces an equivalent set of formulas. Formulas are represented as clauses or lists of literals, implicitly disjoined.

The heart of the simplifier is a term rewrite system, here called “the rewriter.” The rewriter takes as input a term and a set of assumptions and returns a term equivalent to the input one, under the assumptions. The rewriter relies primarily upon **REWRITE** rules derived from axioms, definitions, and previously proved lemmas. See page 277 for details on the form and use of **REWRITE** rules. For example, the theorem

```
(IMPLIES (PROPERP X)
          (EQUAL (REVERSE (REVERSE X)) X))
```

is used to replace every term of the form **(REVERSE (REVERSE x))** by **x**, provided the rewriter can establish **(PROPERP x)**. To establish a hypothesis, the rewriter rewrites it under the current assumptions and asks whether the result is non-**F**. Of course, a term among the current assumptions will rewrite to non-**F**.

²⁰In [1] Larry Akers presents novel ideas for specifying and calculating the types of Lisp formulas.

Similarly, the definitions

```
(NOT X) = (IF X F T)
```

and

```
(NLISTP X) = (NOT (LISTP X))
```

are used to rewrite every occurrence of **(NLISTP x)** to **(IF (LISTP x) F T)**.

The rewriter also uses **META** rules which are, in essence, user-defined simplifiers for the logic, coded as functions on s-expressions in the logic. It is possible to state, within the logic, a formula establishing the soundness of such a “metafunction.” Once a metafunction has been proved sound, the theorem prover can be made to apply it as part of the simplifier just as it would a hand-coded Lisp procedure. This provides what is known as *metatheoretic extensibility*.

For example, the user might define the function **CANCEL** so that when given a list structure representing an equation, such as

```
'(EQUAL (PLUS A (PLUS B (PLUS D E)))
  (PLUS B (PLUS C D))),
```

it returns the list structure representing an equivalent equation, such as

```
'(EQUAL (PLUS A E) (FIX C)).
```

After defining **CANCEL** the user may then prove it correct. The appropriate theorem is

```
(EQUAL (EVAL$ T X A)
  (EVAL$ T (CANCEL X) A)).
```

If this theorem is classified as a **META** rule for the function symbol **EQUAL** (see page 289), then the rewriter will apply the function **CANCEL** to every equation it encounters and use the output (if different from the input) as the rewritten term.

To simplify a clause the simplifier rewrites each literal under the assumption that the remaining literals are false. In a sense, the simplifier replaces each literal by its rewritten counterpart. However, there are three special cases. If the rewritten literal is **T**, the clause is a theorem since one of its disjuncts is true. If the rewritten literal is **F**, the literal can be dropped from the disjunction. Finally, if the rewritten literal contains an **IF**-expression, the answer clause is split into two clauses, according to the test of the **IF**.

Below we give an extended example of the clause-level activity of the simplifier. Suppose the simplifier is given the clause {**p q r**}. Attention is initially focussed on the first literal. Let us denote this state of affairs as

$$\{\mathbf{p} \ \mathbf{q} \ \mathbf{r}\}.$$

$$\Uparrow$$

That is, \mathbf{p} is rewritten under the assumption that both \mathbf{q} and \mathbf{r} are false. Suppose \mathbf{p} rewrites to \mathbf{p}' under these assumptions. Then the simplifier state becomes

$$\{\mathbf{p}' \ \mathbf{q} \ \mathbf{r}\},$$

$$\Uparrow$$

and \mathbf{q} is rewritten under the assumption that both \mathbf{p}' and \mathbf{r} are false. Suppose \mathbf{q} rewrites to $(\mathbf{q}' \ (\mathbf{IF} \ \mathbf{t} \ \mathbf{a} \ \mathbf{b}))$. The \mathbf{IF} is split out and the simplifier state becomes

$$\{\mathbf{p}' \ \sim \mathbf{t} \ (\mathbf{q}' \ \mathbf{a}) \ \mathbf{r}\}$$

$$\Uparrow$$

$$\wedge$$

$$\{\mathbf{p}' \ \mathbf{t} \ (\mathbf{q}' \ \mathbf{b}) \ \mathbf{r}\}$$

$$\Uparrow$$

That is, there are now two clauses and the simplification process continues in both. In the first, \mathbf{r} is rewritten, say to \mathbf{r}' , under the assumptions that \mathbf{p}' and $(\mathbf{q}' \ \mathbf{a})$ are false and that \mathbf{t} is true. In the second, \mathbf{r} is rewritten, say to \mathbf{r}'' , under the assumptions that \mathbf{p}' , \mathbf{t} , and $(\mathbf{q}' \ \mathbf{b})$ are false.

The final answer returned by the simplifier is

$$\{\mathbf{p}' \ \sim \mathbf{t} \ (\mathbf{q}' \ \mathbf{a}) \ \mathbf{r}'\}$$

$$\wedge$$

$$\{\mathbf{p}' \ \mathbf{t} \ (\mathbf{q}' \ \mathbf{b}) \ \mathbf{r}''\}$$

The simplifier and rewriter are vastly more complicated than indicated here.²¹ Before starting the scan through the literals, the simplifier determines whether the clause is a propositional tautology or is a consequence of linear arithmetic. Before conditional rewrite rules are tried, the simplifier makes one pass in which it uses only unconditional rules that do not introduce duplications of subterms—such rules are called *abbreviations* in the theorem prover's output. During rewriting, the equality hypotheses among the assumptions are used in a way that will decide whether the formula is a consequence of equality reasoning

²¹In [87] Myung Kim describes an extensive investigation of the idea of using examples to help control the search process of the rewriter.

alone. When the rewriter encounters a **LESSP**-expression, the rewriter uses the linear arithmetic procedure to attempt to decide whether the **LESSP** relation follows by linear arithmetic reasoning from the assumptions. Finally, the rewriter may actually add some assumptions to those supplied in the top-level call, and its answer is equivalent to the input only under the augmented assumptions. These additional assumptions must be accounted for when the answer clauses are formed after rewriting each literal. A more or less complete description of the simplifier can be found by reading Chapters VII through IX of [24], then reading how metafunctions were added in [25], and finally reading of the integration of linear arithmetic in [31].

10.4. Elimination of Destructors

The destructor elimination process attempts to trade “destructive” terms for “constructive” ones, using rules derived from axioms and previously proved lemmas.

For example, suppose we wish to prove a formula of the form $(p\ x\ (SUB1\ x))$. Suppose x is known to be a positive integer. It can then be represented by $(ADD1\ I)$, for some natural number I . But then $(SUB1\ x)$ is I . Thus we can transform $(p\ x\ (SUB1\ x))$ into $(p\ (ADD1\ I)\ I)$, trading the “destructive” term $(SUB1\ x)$ for the “constructive” term $(ADD1\ I)$.

If x is not known to be a positive integer we can split the goal formula into two subgoals on that condition. In the case where x is known not to be a positive integer the expression $(SUB1\ x)$ will simplify to 0.

Such trading of **SUB1** for **ADD1** is actually carried out in two steps. First, we use the **ELIM** rule generated from the theorem (axiom)

$$\begin{aligned} &(\text{IMPLIES } (\text{NOT } (\text{ZEROP } x)) \\ &\quad (\text{EQUAL } (\text{ADD1 } (\text{SUB1 } x))\ x)), \end{aligned}$$

to replace x by $(ADD1\ (SUB1\ x))$. Thus, we transform $(p\ x\ (SUB1\ x))$ into $(p\ (ADD1\ (SUB1\ x))\ (SUB1\ x))$. Second, we generalize $(SUB1\ x)$ to I , constraining I to be a **NUMBERP** since we know (from type set considerations) that $(SUB1\ x)$ is numeric.

Note that it is the **ELIM** rule that determines which terms are “destructive” and which are “constructive.” The user could arrange to eliminate **ADD1**s in favor of **SUB1**s.

There are much more sophisticated uses of the destructor elimination technique. A very useful application in number theory appeals to the fact that any number x can be represented as $R + YQ$, where R and Q are the remainder and quotient, respectively, of x by some non-0 number Y . Thus, given

(p X (REMAINDER X Y) (QUOTIENT X Y)),

where **X** is known to be a **NUMBERP** and **Y** is a positive **NUMBERP**, the destructor eliminator process could produce the goal

(p R+YQ R Q)

where **R** and **Q** are **NUMBERPs** and **R** is less than **Y**.

The first step is carried out by the **ELIM** rule generated from the theorem

(IMPLIES (AND (NUMBERP X)
 (NOT (ZEROP Y)))
 (EQUAL (PLUS (REMAINDER X Y)
 (TIMES Y (QUOTIENT X Y)))
 X)).

The second is carried out by the **GENERALIZE** rule generated from

(IMPLIES (NOT (ZEROP Y))
 (LESSP (REMAINDER X Y) Y))

together with the observation that **REMAINDER** and **QUOTIENT** are numerically valued.

The above mentioned **ELIM** lemma essentially declares **REMAINDER** and **QUOTIENT** to be “destructive” and **PLUS** and **TIMES** to be “constructive.” Trading one set of symbols for the other is productive only if the data base contains a useful set of rules for manipulating the new symbols. It is certainly the case that the theorem prover finds it easier to cope with **ADD1**, **PLUS**, and **TIMES** than their destructive counterparts **SUB1**, **DIFFERENCE**, and **REMAINDER** and **QUOTIENT**. However, we offer no objective characterization of “destructive” nor do we explain why “constructive” functions better suit the theorem prover’s heuristics. See page 293 for details on the form and use of destructor elimination rules.

10.5. Heuristic Use of Equalities

A common way to use an inductive hypothesis that is an equality is to substitute one side of it for selected occurrences of the other side elsewhere in the conjecture and then to throw away the hypothesis. We call this *cross-fertilization*. This heuristic is implemented in the cross-fertilization process.

Throwing away a hypothesis is a way of generalizing the conjecture being proved. It is thus possible for the cross-fertilization process to take a theorem as input and produce a non-theorem as output. That is, cross-fertilization can cause the system to adopt a non-theorem as its subgoal—a heuristic mistake since the

subgoal cannot be established. We try to avoid this mistake by throwing away the equality hypothesis only if we have already done an induction as a prior step in the proof attempt. Most often, the equality hypothesis thus thrown away is actually an inductive hypothesis.

No user-supplied rules affect the cross-fertilization process.

There is an interaction between cross-fertilization and simplification that, when it occurs, confuses many users. We comment upon it here.

The fertilization process is not the only process in the theorem prover that uses equality hypotheses to rewrite formulas. The simplifier also uses those equalities to canonicalize the formula, replacing all occurrences of the “complicated” side by the “simpler” side. Sometimes the rewriting performed by the fertilization process undoes that done by the simplifier. This momentarily produces a formula that is no longer in canonical form. When, in addition, the equality hypothesis is not thrown away by fertilization, the next simplification will undo the effects of the fertilization and a loop threatens. The loop is avoided by an explicit check in fertilization.

However, it is sometimes the case that the user has anticipated the production of the ephemeral uncanonical formula and has provided a rewrite rule explicitly to handle it. Unfortunately, that rewrite rule will not be applied because the formula is canonicalized by the simplifier from the inside-out. The solution to this problem is to use what we call “bridge” lemmas (see page 406).

10.6. Generalization

The generalization process replaces certain nonvariable terms by variables. For example, the system might generalize

```
(EQUAL (APPEND (REVERSE A) NIL)
      (REVERSE A))
```

by replacing **(REVERSE A)** by the new variable **L**:

```
(EQUAL (APPEND L NIL)
      L).
```

Obviously, if the output formula is a theorem, then the input formula is also a theorem (by instantiation). It is sometimes necessary to so strengthen a conjecture before attempting an inductive proof. However, the resulting generalization may be “too strong” in that it may fail to be a theorem, even though the input formula is a theorem.

To help prevent this, the generalization process constrains the variables introduced by taking note of the **GENERALIZE** rules (see page 294). For example, if a **GENERALIZE** rule has been produced from the theorem **(PROPERP**

(**REVERSE X**) then the subsequent generalization of (**REVERSE A**) to **L** will include the hypothesis that **L** is **PROPERP**. That is, the generalization of

```
(EQUAL (APPEND (REVERSE A) NIL)
      (REVERSE A))
```

will be

```
(IMPLIES (PROPERP L)
      (EQUAL (APPEND L NIL)
            L)).
```

The generalization process also takes into account the type set of the term being replaced. If the type set is a singleton, {**r**}, then the new variable is restricted to the shell **r**.

Recall that the destructor elimination process is actually carried out in two steps, the second step being the introduction of new variables for the destructor terms. It is actually the generalization mechanism—including the use of **GENERALIZE** rules and type sets—that is used.

10.7. Elimination of Irrelevancy

Irrelevant hypotheses in a conjecture may make it difficult to prove the conjecture by induction. The system has some elementary heuristics for identifying irrelevant hypotheses. Roughly speaking, it partitions the literals of the input clause into cliques whose variable symbols do not overlap and then deletes those literals in cliques that can, apparently, be falsified by instantiation. We say “apparently” because it does not actually choose instantiations but rather uses type set reasoning and several heuristics.

10.8. Induction

The system attempts to find inductive arguments suitable for a conjecture by analyzing the recursive functions called in it. Each recursion “suggests” a corresponding induction if the “measured” argument positions are occupied by variable symbols. For example, the term (**APPEND A B**) suggests induction on **A** by successive **CDRs** with the base case (**NLISTP A**), because **APPEND** recurses in that way on its first argument. The term does not suggest induction on **B**. The term (**APPEND (CAR A) B**) does not suggest any induction: the term in the first argument position, (**CAR A**), is not a variable and so we cannot form an induction hypothesis by instantiating it as in the recursion; the

term in the second argument position is irrelevant because **APPEND** does not recurse into it.

The analysis of which argument positions are important takes place when functions are defined. Recall admissibility requirement (d) of the principle of definition. It requires that some measure **m** of the arguments decrease according to **ORD-LESSP** in each recursive call. When a definition is admitted by the mechanized definitional principle, a suitable **m** is found (or provided by the user). It is frequently the case that **m** does not measure all of the arguments of the definition. For example, the measure term traditionally used to justify the definition of (**APPEND X Y**) is (**COUNT X**). The arguments actually used in **m** are the ones considered measured for purposes of the inductive analysis. We call those arguments a *measured subset* of the formals of the defined function.

Many functions have more than one measured subset. For example, suppose (**LT I J**) is defined to decrement both **I** and **J** by **SUB1** until either is **ZEROP**. Then both the **COUNT** of **I** and the **COUNT** of **J** decrease in each recursion. Either is sufficient to admit **LT**. But if both are known to the theorem prover, then (**LT I J**) suggests both induction on **I** and induction on **J**. Furthermore, a term such as (**LESSP (CAR X) J**) suggests induction on **J** even though the first argument is “blocked” by a nonvariable.

Thus, the system’s heuristics for choosing an induction suitable for a conjecture are more powerful if the system has alternative measured subsets for each function involved. By proper use of the **DEFN** command, and in particular, use of the **hints** argument, the user can point out alternative justifications. This is the only way the user can influence the induction process—except to override it completely using the **hints** argument of **PROVE-LEMMA**.

Once a set of suggested induction principles is identified, the system merges them to try to accommodate the various functions involved. Finally, it chooses between the final candidates using a variety of heuristics. No user-supplied rules influence these latter moves.

A complete description of the induction mechanism can be found in [24] with the minor addition described in [29].

11 The Four Classes of Rules Generated from Lemmas

As previously explained, the data base contains rules which determine the behavior of the system. Each rule has a name, and whenever the rule is used by the system the name of the rule is printed.²² Each rule has a *status* which is either **ENABLED** or **DISABLED**. When the status of a rule is **DISABLED** the rule is not used.

There are four different *classes* of rules:²³

REWRITE,
META,
ELIM, and
GENERALIZE.

Each class of rule influences a different aspect of the theorem prover's behavior. For example, a **REWRITE** rule may direct the system to replace all instances of

²²This is not strictly true. See the discussion of "immediate dependency" on page 319.

²³Historical Note: In the version of the theorem prover described in *A Computational Logic* we had another class of rules, the so-called **INDUCTION** rules. In that version of our system, the **DEFN** command attempted to construct a suitable measure for each submitted definition. The **INDUCTION** rules were used to guide this construction. **DEFN** no longer tries to construct a measure; it uses **COUNT** unless told to do otherwise by the user. Thus, **INDUCTION** rules have been eliminated.

one term by corresponding instances of another, provided certain conditions are satisfied. To use the theorem prover effectively, it is necessary to understand how such rules influence the behavior of the system.

However, the user of the theorem prover does not deal explicitly with rules; instead, the user deals with formulas in the logic, and the rules themselves are generated from the formulas. The name of each generated rule is the name of the formula from which the rules were generated. For example, a rule to replace all occurrences of (instances of) **(REVERSE (REVERSE X))** by **X** is generated from the formula **(EQUAL (REVERSE (REVERSE X)) X)**. To use the theorem prover effectively it is thus also necessary to understand how rules are generated from formulas: once the need for a given rule is recognized, the user must find a formula that will generate it.

Finally, our system is a theorem prover, not just a formula manipulator. Thus the transformations effected by the rules are supposed to be valid inferences in the logic. It turns out that the rules generated from a formula are valid iff the formula is a theorem. For example, the rule that replaces **(REVERSE (REVERSE X))** by **X** is not valid since **(EQUAL (REVERSE (REVERSE X)) X)** is not a theorem. The effort to prove a formula so as to obtain a certain rule often reveals that the desired rule is invalid.

The whole rule-validation-and-generation process is brought together at the user interface in a deceptively simple way: when the user submits a formula to the theorem prover, it must be tagged with a set of tokens taken from **{REWRITE META ELIM GENERALIZE}**; if the formula is proved the corresponding rules are generated and added.

This interface is in many ways *too* simple; the user is vitally concerned with the production of rules but can discuss only formulas. This has two bad side effects. First, elegant and often simple truths are obscured by the need to express them in a way that permits their transformation into rules.²⁴ Second, the absence of explicit discussion of rules often lulls the inexperienced user into an ill-considered disregard for the procedural interpretation of his or her lemmas. This almost always comes back to haunt the user; halfway through the construction of his or her first big proof, the user finds the system acting as though it were programmed by a monkey: obvious inferences are missed, trivial proofs are unimaginably difficult to obtain, and the system frequently takes (literally) forever to respond. The problem is that the data base is a spaghetti bowl of conflicting rules. One solution is to use **DISABLE** commands to turn off virtually every rule and then use the hint facility to tell the system every proof.

²⁴A good solution to this problem is to prove the elegant form of the theorem as a lemma with the empty set of rule classes. This will store the formula under its name but generate no rules. Then prove the obscure, rule-generating version by explicit appeal (via a **USE** hint, see page 364) to the elegant formulation.

Resist this urge! If you begin to fight the system, by constantly giving it tactical advice, you probably have not adequately explicated a strategy for handling the concepts. Once you begin to fight it, you must fight it forever; *its ability to tolerate detail and keep slogging works against you* because it keeps throwing out problems for you to solve. It is thus better to tell it how to solve the problems. Read its proof attempts as it goes, stop it as soon as it has gotten off the “right” path, and explain a strategy by proving rules. This harnesses its tolerance for detail to work in your favor; it will apply your strategy tirelessly.

In summary, to harness the system you must understand

- how rules are used by the system,
- how rules are generated from formulas, and
- how formulas are proved.

The third issue, of course, encompasses this whole handbook.

We discuss the other two issues at length in this chapter. We deal with each of the two issues separately for each of the four types of rules, **REWRITE**, **META**, **ELIM**, and **GENERALIZE**. Within each section we first describe how rules of the type are used by the system—without regard for the logical validity of the transformations effected. Then we discuss how such rules are derived from formulas.

11.1. REWRITE

REWRITE rules are by far the most common, the most useful, and the most complicated kind of rule used by the system.

There are four different forms of **REWRITE** rules:

- **Type Prescription Rules:** used throughout the theorem prover to quickly compute the possible shell types returned by an expression,
- **Compound Recognizer Rules:** used to forward chain from user-defined predicates to conjunctions and disjunctions of shell types,
- **Linear Arithmetic Rules:** used inside the simplifier by the linear arithmetic decision procedure and its heuristic extension, and
- **Replacement Rules:** used inside the simplifier by the term rewrite system, to replace “complicated” terms by “simpler” ones.

Which kind of **REWRITE** rule is generated from a given formula depends on the syntactic form of the formula. In fact, we first transform the formula, introducing explicit constants in place of certain **IDENTITY** terms and flat-

tening the formula into a set of conjuncts. Then we make a rule for each of the conjuncts. (We make this more precise below.) Thus, several rules of each kind may be generated from a single submitted formula.

Given a conjunct, we generate a type prescription rule, if possible. If a given conjunct cannot be encoded as a type prescription, we try to generate a compound recognizer rule. If that fails, we try to generate a linear arithmetic rule. If that fails, we try to generate a replacement rule. Sometimes it is impossible to generate any rule for a conjunct and this is reported. Despite the priorities above, almost all **REWRITE** rules are replacement rules. We therefore describe the form and meaning of replacement rules first and discuss the other types of rules later. But first we explain how we transform the formula into a set of conjuncts.

The first transformation we make replaces terms of the form (**IDENTITY term**) by **val**, where **term** is a variable-free term that reduces by computation to the explicit value **val**. This transformation essentially allows the abbreviation of constants. For example, if one's rewrite strategy calls for a rule involving (**SUM A B 4294967296**) then it is possible to write (**SUM A B (IDENTITY (EXPT 2 32))**), provided **EXPT** is defined as the exponentiation function. The latter is more perspicuous. Observe that (**SUM A B (EXPT 2 32)**) will not work here if the intention is that the term match (**SUM A B 4294967296**). **IDENTITY** is afforded this special treatment only here, in the preprocessing of **REWRITE** rules, and in similar preprocessing of **EXPAND** hints (see page 365). Otherwise, **IDENTITY** is given no special treatment.

The second transformation flattens the **AND** and **IMPLIES** structure of the formula, transforming the submitted formula into an equivalent conjunction over a set of formulas, each of the form

$$(\text{IMPLIES } (\text{AND } h_1 \dots h_n) \text{ concl})$$

where no h_i is a conjunction and **concl** is neither a conjunction nor an implication. For example, the formula

$$\begin{aligned} &(\text{AND } (\text{IMPLIES } h_1 \\ &\quad (\text{AND } \text{concl}_1 (\text{IMPLIES } h_2 \text{ concl}_2))) \\ &\quad (\text{IMPLIES } (\text{AND } h_3 h_4) \\ &\quad (\text{IMPLIES } h_5 \text{ concl}_3))) \end{aligned}$$

is transformed into a conjunction of the following three formulas:

$$\begin{aligned} &(\text{IMPLIES } h_1 \text{ concl}_1) \\ &(\text{IMPLIES } (\text{AND } h_1 h_2) \text{ concl}_2) \\ &(\text{IMPLIES } (\text{AND } h_3 h_4 h_5) \text{ concl}_3) \end{aligned}$$

After removing the described calls of **IDENTITY** and so flattening the **AND** and **IMPLIES** structure of the formula, we generate a type prescription, a compound recognizer, a linear arithmetic, or a replacement rule from each of the resulting formulas.

We therefore consider how we generate one of the four kinds of rules from a formula of the form (**IMPLIES** (**AND** $h_1 \dots h_n$) **concl**).

11.1.1. Replacement Rules

Replacement rules are the most common form of **REWRITE** rules. They permit the rewriter to simplify terms by replacing subterms with equivalent ones. For example, a replacement rule might direct the system to “replace terms of the form (**REVERSE** (**REVERSE** x)) by x , provided (**PROPERP** x) can be established.” Another is “In the tests of **IF**s and in other propositional occurrences, (**ASSOC** x (**PAIRLIST** u v)) should be replaced by (**MEMBER** x u).” Note that in the second example the two terms are not **EQUAL**: the **ASSOC** returns a **CONS**-pair when the **MEMBER** returns **T**. But the two terms are propositionally equivalent—they return **F** or non-**F** in unison—and hence are interchangeable in contexts where they are being treated as propositions.

In general, a *replacement rule* is a rule of the form **Replace** (or **Propositionally replace**) **lhs** by **rhs** after establishing **hyp**₁, ..., **hyp**_n, where **lhs** (*left-hand side*), **rhs** (*right-hand side*) and **hyp**₁, ..., **hyp**_n (*hypotheses*) are all terms.

Replacement rules are applied by the rewriter in an inside-out fashion. That is, replacement rules are applied to (**fn** $a_1 \dots a_n$) only after the a_i have been rewritten.

To apply replacement rules to the target term (**fn** $a_1 \dots a_n$) the rewriter searches the enabled rules in reverse chronological order, i.e., starting with the most recently proved one, looking for the first applicable rule. A rule with left-hand side **lhs**, right-hand side **rhs** and hypotheses **hyp**_{*i*}, *i*=1, ..., *n*, is *applicable* to the target term iff all of the following conditions are met:

- the rule is of the “**Replace**” form; or the rule is of the “**Propositionally replace**” form and the target occurs as the test of an **IF**, a literal in a clause, or a hypothesis of a rule;
- there is a substitution σ such that **lhs**/ σ is syntactically identical to the target;
- the hypotheses of the rule can be relieved under σ ; and
- certain heuristic conditions are met.

The heuristic conditions prevent certain implausible backchaining paths and some infinite loops. (We discuss the implications of some of these heuristics later.) The test for applicability yields a substitution, σ' , which is usually just σ but is, in general, an extension of σ . (We discuss how the extension is constructed later.)

When the first applicable rule is found, with substitution σ' , the rule is applied by replacing the target with **rhs**/ σ' and recursively rewriting the result.

To *relieve the hypotheses* of a rule, under some substitution σ , the rewriter iteratively considers the hypotheses in the order in which they are presented in the rule, attempting to establish each in turn and possibly extending σ each iteration. Intuitively, a hypothesis **hyp₁** is established under σ iff either **hyp₁**/ σ is among the current assumptions or recursively rewrites to **T**. However, it is possible that some **hyp₁** contains variables not bound by σ . Such variables are called *free variables*. To establish such a hypothesis under σ , the rewriter attempts to find an extension σ' of σ such that **hyp₁**/ σ' is among the current assumptions (or the “ground units,” i.e., variable-free unconditional replacement rules). One special case is recognized: if **hyp₁**/ σ is of the form (**EQUAL v term**), where **v** is a variable not occurring in **term**, a suitable extension is to substitute the rewritten form of **term** for **v**. If any suitable extension can be found, the rewriter extends σ with the first extension found and moves on to the next hypothesis. It does not backtrack to try alternative extensions.

The heuristic conditions are present to prevent some forms of infinite looping and implausible backwards chaining. The most commonly felt restriction pertains to rules that express commutativity or, in general, rules that permit one to permute the arguments in a nest of functions. For example, were the rule “**Replace (PLUS X Y) by (PLUS Y X)**” applicable to all expressions matching the left-hand side, the rewriter would loop forever, rewriting (**PLUS A B**) to (**PLUS B A**) and then to (**PLUS A B**) and then to (**PLUS B A**), etc. Such “permutative” rules are applied only if **rhs**/ σ occurs earlier than **lhs**/ σ in a lexicographically based enumeration of terms. Thus, using the commutativity of **PLUS** rule, (**PLUS B A**) rewrites to (**PLUS A B**) but not vice versa. Similarly, the rule “**Replace (DELETE X (DELETE Y L)) by (DELETE Y (DELETE X L))**” is applicable to (**DELETE B (DELETE A LST)**) and rewrites it to (**DELETE A (DELETE B LST)**); but the rule is not applicable to (**DELETE A (DELETE B LST)**).

This completes our explanation of how replacement rules are used by the theorem prover. On page 395 we give some hints for the effective use of such rules. We next consider how replacement rules are generated from formulas.

Suppose a formula of the form (**IMPLIES (AND hyp₁ ... hyp_n) concl**) is submitted and proved in class **REWRITE**. If the formula generates a type prescription, compound recognizer or linear arithmetic rule, no replacement rule is generated. We discuss these three forms of rules later. If **concl** is a

variable symbol or **QUOTED** constant, the rule is not a legal **REWRITE** rule and an error is caused. If **concl** is of the form **(EQUAL lhs rhs)**, we generate the replacement rule

Replace lhs by rhs
after establishing hyp₁, ..., hyp_n.

If **concl** is of the form **(IFF lhs rhs)** we generate

Propositionally replace lhs by rhs
after establishing hyp₁, ..., hyp_n.

If **concl** is of the form **(NOT lhs)**, it is treated as though it were **(EQUAL lhs F)**. A conclusion of any other form is treated as though it were **(IFF concl T)**.

The function definition

Definition.

(fn x₁ ... x_n) = body

may be thought of as generating the replacement rule

Replace (fn x₁ ... x_n) by body,

in the sense that definitions, when used, are used only to replace function calls by their instantiated bodies. We call this “opening up” or “expanding” the definition. *Definitions are never “folded” or used “backwards.”*

However, the applicability heuristics are quite specialized for replacement rules generated from definitions. For definitions that are not explicitly recursive, that is, for those definitions in which **fn** is not used as a function symbol within **body**, the applicability heuristics are almost the same as for unconditional replacement rules, except that the theorem prover monitors the number of **IF** expressions introduced and the size of the arguments that are duplicated by the expansion. These constraints are manipulated so that the simplifier will eventually expand every nonrecursive function application in a formula, though they may not all expand in a single call of the rewriter.

For definitions that are explicitly recursive, the applicability conditions are quite different than for unconditional replacements. There are several conditions under which we will expand a recursive definition, but the most commonly occurring one is that all of the argument expressions in the recursive calls in the rewritten function body are subterms of the current conjecture. Thus **(APPEND X Y)** will expand and introduce **(APPEND (CDR X) Y)** provided **(CDR X)** and **Y** are already in the current conjecture. If **(CDDR X)** is in the conjecture, **APPEND** will expand again. If not, it will not. This heuristic works well with induction: the induction hypothesis is exactly the information the rewriter needs to know which functions should be expanded.

11.1.2. Type Prescription Rules

A type prescription rule allows the system to compute the “type set” of an expression. Because of the ubiquity of type sets it is important that they be implemented efficiently. In particular, we desire a fast way to compute a type set for a term under a set of type set assumptions about other terms. Of course, any given term has many type sets, namely, every extension of the smallest one. Determining the smallest type set of a term is an undecidable problem, since the term is a theorem iff **FALSEP** is not in its smallest type set. Thus, we must content ourselves with an algorithm for computing some type set. Type prescription rules are a way that the user of the theorem prover can influence the type set algorithm.

Suppose the following formula were a theorem:

```
(OR (LISTP (NORM X Y))
    (LITATOM (NORM X Y))
    (EQUAL (NORM X Y) Y)).
```

One way this theorem could be used is to determine a type set for **(NORM a b)**, namely the union of **{LISTP LITATOM}** and the type set of **b**. A type prescription rule is an encoding of the information in such formulas.

A *type prescription rule* associated with a function symbol **fn** of arity **n** is an **n+1**-tuple **<ts, flg₁, ..., flg_n>** where **ts** is a type set and each **flg_i** is either true or false. The effect of such a rule, when enabled, is that a type set of a term **(fn a₁ ... a_n)** is the union of **ts** together with the union of type sets of those **a_i** such that **flg_i** is true. The most recently proved rule is used.

We say a term **t** is a *type set term* about a term **a** if **t** either has the form

```
(OR (r1 a)
    ...
    (rn a))
```

or the form

```
(AND (NOT (r1 a))
    ...
    (NOT (rn a)))
```

where each **r_i** is a recognizer.

The *type set described* by a type set term **t** is either **{r₁ ... r_n}** or **Ω-{r₁ ... r_n}**, depending on whether **t** is a type set term of the first or second form above.

A formula is a *type prescription formula* if it has the form

```
(OR tst
  (equal (fn v1 ... vn) var1)
  ...
  (equal (fn v1 ... vn) vark)),
```

where the v_i s are distinct variable symbols and each var_m is some v_i and tst is a type set term about $(fn\ v_1\ \dots\ v_n)$. The type prescription rule *generated* from such a formula is $\langle ts, flg_1, \dots, flg_n \rangle$, where ts is the type set described by tst and flg_i is true iff one of the disjuncts in the formula is the equation $(EQUAL\ (fn\ v_1\ \dots\ v_n)\ v_i)$. The rule is associated with the function symbol fn .

If a formula is submitted as class **REWRITE** and it satisfies the requirements of a type restriction formula, we store the generated type prescription rule in the data base.

We actually recognize a wider class of formulas than merely those having the syntactic forms described. In particular, a special-purpose program tries to put the submitted formula into type prescription form using

- the definitions of nonrecursively defined Ground Zero functions (e.g., **IMPLIES** and **NLISTP**);
- the fact that $(EQUAL\ X\ (const))$ is equivalent to $(r\ X)$ if **const** is a shell constructor of no arguments and no base function (or **TRUE** or **FALSE**) and r is the corresponding recognizer (or **TRUEP** or **FALSEP**, respectively);
- the fact that recognizers are all mutually exclusive; and
- propositional calculus with equality.

Table 11.1 contains some examples of formulas submitted as **REWRITE** formulas that are recognized as type prescription terms and their generated type prescription rules.

Table 11.1

formula	type prescription
<pre>(OR (LISTP (FN X Y)) (NUMBERP (FN X Y)) (EQUAL (FN X Y) Y))</pre>	$\langle \{ \text{LISTP NUMBERP} \}, \text{F T} \rangle$
<pre>(IMPLIES (LISTP (FN X Y)) (EQUAL (FN X Y) Y))</pre>	$\langle \Omega - \{ \text{LISTP} \}, \text{F T} \rangle$
<pre>(IMPLIES (NOT (EQUAL (FN X Y) T)) (EQUAL (FN X Y) F))</pre>	$\langle \{ \text{TRUEP FALSEP} \}, \text{F F} \rangle$
<pre>(IMPLIES (AND (LISTP (FN X Y)) (NOT (NUMBERP (FN X Y)))) (EQUAL (FN X Y) Y))</pre>	$\langle \Omega - \{ \text{LISTP} \}, \text{F T} \rangle$

11.1.3. Compound Recognizer Rules

A compound recognizer rule allows the system to “forward chain” from a user-defined predicate about a term to type set information about the term. For example, suppose we defined a recursive function, **FORMP**, that recognized s-expressions composed of literal atoms and proper lists of **FORMPs** beginning with literal atoms. Thus

```
(FORMP '(PLUS X (ADD1 Y))) = T
```

```
(FORMP 'X) = T
```

```
(FORMP 3) = F
```

```
(FORMP '(PLUS *1*TRUE X)) = F
```

Observe that if **x** is a **FORMP** then **x** is either a **LITATOM** or a **LISTP**. It is sometimes convenient to be able to make such observations. For example, if **x** is known to be a non-**LISTP** and to be a **FORMP**, then it is clear that **x** must be a **LITATOM**. Such observations could be made by expanding the definition of **FORMP** and simplifying. But that means that the relationship between **FORMP** and the type mechanism must be rediscovered every time it is to be used.

FORMP is an example of what we call a “compound recognizer”—a predicate that, in a weak sense, acts as a recognizer for a type set. A theorem such as

```
(IMPLIES (FORMP X)
  (OR (LITATOM X)
    (LISTP X)))
```

can be used to forward chain from the hypothesis that **(FORMP x)** is true to the information that the type set of **x** is a subset of **{LITATOM LISTP}**.

What can we infer from the hypothesis that **(FORMP x)** is false? Certainly, we do not know that **x** is neither a **LITATOM** nor a **LISTP**—**x** could be a **LISTP** that violates the rules of **FORMP** on some element. But we do know that **x** is not a **LITATOM**, because all **LITATOMs** are **FORMPs**. In general, the type information gleaned from the assumption that **(fn x)** is true is different from that gleaned from the assumption that **(fn x)** is false.

A *compound recognizer rule* is a triple of the form **<parity, fn, ts>** where **parity** is either **TRUE** or **FALSE**, **fn** is a function symbol of arity 1 and **ts** is a type set. If we have such a rule, we say **fn** is a *compound recognizer of parity parity for type set ts*.

The rewriter uses compound recognizer rules as follows. Recall that the assumptions made during rewriting are recorded by assigning type sets to terms. For example, assuming **(NUMBERP x)** true actually leads to the internal assumption that the argument term, **x**, has type set **{NUMBERP}**. More generally, if **r** is a recognizer, then assuming **(r x)** true sets the type set of **x** to **{r}**. Similarly, if **fn** is a compound recognizer of parity **TRUE** for type set **ts**, then when **(fn x)** is assumed true, the rewriter assigns to **x** the type set obtained by intersecting its current type set with **ts**. This assignment is *in addition to* the normal processing of the assumption. For example, if **fn** is a Boolean valued function, the assumption that **(fn x)** is true will also lead to the assignment of the type set **{TRUE}** to **(fn x)**. The rules of parity **FALSE** are used exactly the same way, when **(fn x)** is assumed false.

A formula is a *compound recognizer formula of parity TRUE* if it has the form

```
(IMPLIES (fn x) tst)
```

or the form

```
(EQUAL (fn x) tst)
```

where **x** is a variable symbol and **tst** is a type set term about **x**. The compound recognizer rule *generated* from such a formula is **<TRUE, fn, ts>**, where **ts** is the type set described by **tst**.

A formula is a *compound recognizer formula of parity FALSE* if it has the form

```
(IMPLIES (NOT (fn x)) tst)
```


or the form

(EQUAL (fn x) tst)

where **x** is a variable symbol and **tst** is a type set term about **x**. If the formula has the first form, then the compound recognizer rule *generated* from the formula is **<FALSE, fn, ts>**, where **ts** is the type set described by **tst**. If the formula has the second form, then the compound recognizer rule *generated* from the formula is **<FALSE, fn, Ω -ts>**, where **ts** is the type set described by **tst**.

If a formula is submitted as class **REWRITE** and it satisfies the requirements of a compound recognizer formula, we store the generated compound recognizer rule(s) in the data base.

Observe that some formulas are compound recognizer formulas of both parities. Such formulas generate two rules. An example is the formula

(EQUAL (BOOLEANP X) (OR (TRUEP X) (FALSEP X))).

When **(BOOLEANP x)** is assumed true, the rewriter intersects the type set of **x** with **{TRUE FALSE}** to get the new type set of **x**. When **(BOOLEANP x)** is assumed false, the rewriter intersects the type set of **x** with **Ω -{TRUE FALSE}** to get the new type set of **x**.

The rule above is a very useful rule about **BOOLEANP** because it completely captures the properties of the predicate in the most efficient way possible.

11.1.4. Linear Arithmetic Rules

By *linear arithmetic* we mean that fragment of natural number arithmetic dealing only with sum and difference, equality, and inequality.

The rewriter has a special purpose procedure for deciding some facts of linear arithmetic. When called upon to rewrite an arithmetic relation, such as a **LESSP** expression, in the context of some assumptions, the rewriter uses the procedure to determine whether the relation is a consequence of the arithmetic relations among the assumptions. It does so by adding the negation of the given relation to the set of assumed relations and attempting to derive an inconsistent ground inequality by cross multiplying and adding inequalities so as to cancel out terms.

For example, suppose that the following two inequalities are among the current assumptions:

(1) **(LESSP (PLUS 3 a) (PLUS b c))**

(2) **(LESSP (PLUS a c) b)**

and suppose we are asked to rewrite **(LESSP a b)**. If a contradiction follows from the assumption **(NOT (LESSP a b))**, then **(LESSP a b)** is equal to

T under the current assumptions. The system thus attempts to derive a contradiction from (1) and (2) and the negation of the relation we are rewriting. (Note that **(NOT (LESSP a b))** is equivalent to **(LESSP b (ADD1 a))** since **LESSP** is the less than relation over the naturals not the rationals.) The linear arithmetic procedure therefore is applied to the following set of inequalities, here written in conventional notation:

$$(1) \quad 3 + a < b + c$$

$$(2) \quad a + c < b$$

$$(3) \quad b < 1 + a$$

The decision procedure eliminates variables one at a time, starting with the lexicographically largest. Suppose that variable is **c**. Adding together (1) and (2) and cancelling **c** produces

$$(4) \quad 3 + 2a < 2b$$

Having eliminated **c** it then eliminates **b**: multiplying each side of (3) by 2 and adding the result to (4) we get

$$(5) \quad 1 < 0$$

which is a contradiction. Thus, the original set of (1)-(3) is unsatisfiable.

Of course, the terms **a**, **b**, and **c** need not be variables. They are, in general, arbitrary (non-**PLUS**, non-**DIFFERENCE**) expressions. Suppose they are in fact

```
a = (LENGTH (CDR STR))
b = (LENGTH (PATTERN K TABLE))
c = (DELTA2 (CDR STR) (PATTERN K TABLE))
```

Nevertheless, the linear arithmetic procedure treats the three addends **a**, **b**, and **c** as though they were variables when cross multiplying and adding inequalities.

However, the fact that the addends are typically nonvariable terms means that often there are interesting arithmetic relationships between them, derivable from previously proved lemmas. For example, assumption (2), above, which now reads

$$(2) \quad (\text{LESSP } (\text{PLUS } (\text{LENGTH } (\text{CDR } \text{STR})) \\ (\text{DELTA2 } (\text{CDR } \text{STR}) \\ (\text{PATTERN K TABLE}))) \\ (\text{LENGTH } (\text{PATTERN K TABLE}))),$$

may be merely a consequence of the more general lemma

Theorem. DELTA2-BOUND:
 (IMPLIES (AND (STRINGP S)
 (STRINGP PAT))
 (LESSP (PLUS (LENGTH S)
 (DELTA2 S PAT))
 (LENGTH PAT)))).

Assumption (2) can be derived from **DELTA2-BOUND** by instantiating **S** to be **(CDR STR)** and **PAT** to be **(PATTERN K TABLE)** and establishing the two **STRINGP** hypotheses. **DELTA2-BOUND** is an example of a **REWRITE** lemma stored as a linear arithmetic rule.

A *linear rule* is a 3-tuple $\langle \text{hyps}, \text{concl}, \text{key} \rangle$, where **hyps** is a list of hypothesis terms implicitly conjoined, **concl** is a term of the form **(LESSP lhs rhs)** or **(NOT (LESSP lhs rhs))**, and **key** is a nonvariable term with the following properties: first, it is one of the addends of **lhs** or **rhs**. Second, it contains all the variables in **concl** except, possibly, those occurring in **hyps**—thus if every variable in **key** and **hyps** is instantiated, every variable in **concl** is also. Third, under some substitutions **key** is the syntactically largest addend in **concl** and thus will be a candidate for immediate elimination.

The linear rule generated from the **DELTA2-BOUND** lemma is

```
<((STRINGP S) (STRINGP PAT)),           ; hyps
  (LESSP (PLUS (LENGTH S) (DELTA2 S PAT)) ; concl
    (LENGTH PAT)),
  (DELTA2 S PAT)>                        ; key
```

Note that the only key addend in the conclusion of **DELTA2-BOUND** is **(DELTA2 S PAT)**.

Roughly speaking, the linear arithmetic procedure uses such rules as follows. When the process of cross multiplying and adding inequalities has terminated and no inconsistency is found, the data base of enabled linear rules is searched. If the key term in some rule can be instantiated to yield the syntactically largest addend in some inequality under consideration, i.e., the addend that must be eliminated next from that inequality, the conclusion of the rule is instantiated and added to the set of inequalities, provided the hypotheses can be relieved by recursive rewriting.

A **REWRITE** lemma of the form **(IMPLIES (AND $h_1 \dots h_n$) concl)** is stored as one or more linear rules, $\langle (h_1 \dots h_n), \text{concl}, \text{key} \rangle$, provided **concl** is a (possibly negated) **LESSP** expression and there exists at least one key addend. A linear rule is generated for every key addend in **concl**.

11.2. META

Recall the example metafunction discussed on page 267. **CANCEL** is defined as a function that takes a list representing an equation, e.g.,

```
'(EQUAL (PLUS A (PLUS B (PLUS D E)))
        (PLUS B (PLUS C D))),
```

and returns a list representing an equivalent, simplified equation, such as

```
'(EQUAL (PLUS A E) (FIX C)).
```

In example file 13 we define **CANCEL** as described above and prove it correct. Readers interested in developing metatheoretic simplifiers of their own are invited to study that example first.

The **META** rule associated with **CANCEL** is “**CANCEL is a metatheoretic simplifier for EQUAL.**” If **CANCEL** were redefined so as to apply not only to equations (i.e., lists that begin with **EQUAL**) but to inequalities represented by **LESSP** expressions, it would be useful to generate the additional rule: “**CANCEL is a metatheoretic simplifier for LESSP.**”

If **simp** and **fn-symb** are function symbols in the logic, then “**simp is a metatheoretic simplifier for fn-symb**” is a **META** rule. We call **fn-symb** the *target function symbol* of the rule. **META** rules are stored among the replacement rules and, like the replacement rules, are tried in reverse chronological order. Roughly speaking, when the rewriter is searching for an applicable **REWRITE** rule for **(fn a₁ ... a_n)** and finds an enabled metatheoretic simplifier, **simp**, for **fn**, it attempts to reduce

```
(simp '(fn a1 ... an))
```

to an explicit value, **v**, by evaluating the definition of **simp**. If successful and **v** is different from **(fn a₁ ... a_n)** and all the variables of **v** are among those of **(fn a₁ ... a_n)**, then the rewriter replaces **(fn a₁ ... a_n)** by **v**.

The description above is inaccurate in two ways. First, metatheoretic simplifiers are not applied to all terms but merely to “tame” ones. Second, the output **v** of the simplifier is used to replace **(fn a₁ ... a_n)** only if **v** is tame. We discuss “tameness” at length below. But before we do we note that a nontame term can be generalized to a tame one by replacing certain offensive subterms by variables. When the system is considering applying a metatheoretic simplifier to a term and the term is found to be nontame, it is first generalized to a tame term, the simplifier is applied to the generalization, and the result, if tame, is then instantiated so as to “invert” the generalization.

The notion of “tameness” is a purely syntactic one, defined below. The key property of tame terms is that if \mathbf{v} is a tame term whose variable symbols are \mathbf{x}_1 , ..., \mathbf{x}_n , then \mathbf{v} is equal to the “evaluation of ' \mathbf{v} ,’” that is, \mathbf{v} is equal to

```
(EVAL$ T 'v
  (LIST (CONS 'x1 x1)
        ...
        (CONS 'xn xn)))
```

More succinctly, if \mathbf{v} is tame it is equal to the evaluation of its quotation, under the standard alist on its variable symbols. This is not necessarily the case for nontame terms.

In order to generate a **META** rule, a formula must have the form

```
(EQUAL (EVAL$ T v a)
  (EVAL$ T (simp v) a)),
```

where **simp** is a function symbol of one argument and \mathbf{v} and \mathbf{a} are distinct variable symbols.

To better understand the role of tameness in this process, suppose \mathbf{v} is a term to which we wish to apply **simp**. If \mathbf{v} is tame, then it is equal to the evaluation of ' \mathbf{v} ' under the standard alist. By the theorem justifying the **META** rule for **simp**, the evaluation of ' \mathbf{v} ' is equal to that of (**simp** ' \mathbf{v} '). But (**simp** ' \mathbf{v} ') can be reduced to an equivalent explicit value ' \mathbf{w} '.²⁵ If \mathbf{w} is itself tame, then the evaluation of ' \mathbf{w} ' is equal to \mathbf{w} . Hence, we are justified in replacing \mathbf{v} by \mathbf{w} .

What do we mean by “tame?” We define it below, along with the mutually recursive concept “total.” Tameness is a syntactic property of terms, while totality is a syntactic property of function symbols. Roughly speaking, a term is tame iff every function in it is total, and a function is total iff its body is tame. However, the presence of **V&C\$** and recursion complicate matters.

A term is *tame* if it is a variable, a constant, the application of a “total” function to tame terms, a term of the form (**V&C\$** T ' \mathbf{t} alist) or (**EVAL\$** T ' \mathbf{t} alist) where \mathbf{t} and **alist** are tame, or a term of the form (**FOR** \mathbf{v} \mathbf{r} '**cond** **op** '**body** alist) where each of \mathbf{v} , \mathbf{r} , **cond**, **op**, **body** and **alist** are tame.

We classify each function symbol as to whether it is total or not at the time it is introduced, as a function of its body and the previously determined totality of its subfunctions.

Intuitively, **fn** is total if its body is tame, which means, roughly, that every function called in the body is total. However, the body may involve recursive

²⁵Possibly not. Metatheoretic simplifiers ought to be explicit value preserving. If they are not, and on some explicit value they “run forever” or involve undefined function calls, time is wasted but no harm results.

calls of **fn** itself. When we are trying to determine whether **fn** is total, do we assume it is while computing the tameness of its body? At first glance the answer seems to be “yes, because we have proved, during the admission of the function, that every recursive call of **fn** decreases a measure of the arguments in a well-founded sense.” But consider calls of **fn** occurring inside of quoted expressions given to **EVAL\$**. Those calls have not been checked. For example, assuming **fn** total and then asking if its body is tame would result in the following function being considered total:

Definition.

(RUSSELL) = (NOT (EVAL\$ T '(RUSSELL) NIL)),

since, if **RUSSELL** is considered total then the **EVAL\$** expression above is tame. Therefore, in the definition of “total” we do not use the notion of “tame” and instead use the notion of “super-tame” which means that **fn** is considered total outside of **EVAL\$** expressions—where the definitional principle ensures termination—but not inside. Here are the precise definitions of “total” and “super-tame.”

All the primitives except **V&C\$**, **V&C-APPLY\$**, **EVAL\$**, **APPLY\$** and **FOR** are *total*. A function defined by **(fn x₁ ... x_n)=body** is *total* iff **body** is super-tame with respect to **fn**.

A term is *super-tame* with respect to **fn** iff it is a tame term (under the assumption that **fn** is not total) or it is a call of a total function or **fn** on super-tame terms.

Every function definable without use of **V&C\$** and its dependents is total. In addition, functions that involve **V&C\$**, **EVAL\$** and **FOR** are total provided the interpreted arguments are the quotations of tame terms and the other functions in the definition are total.

As noted above, metafunctions can only be applied to the quotations of tame terms, but any term can be generalized to a tame term. We are now in a position to illustrate this claim. The equation

**(EQUAL (PLUS B (PLUS C (RUSSELL)))
(PLUS A (PLUS B D)))**

is not tame; but the equation

**(EQUAL (PLUS B (PLUS C X))
(PLUS A (PLUS B D)))**

is tame and is a generalization of the original equation. The **CANCEL** metafunction can be applied to the quotation of the latter equation and produces the quotation of

**(EQUAL (PLUS C X)
(PLUS A D)).**

By applying the inverse of the generalizing substitution we instantiate this to

```
(EQUAL (PLUS C (RUSSELL))
        (PLUS A D)).
```

By the correctness of **CANCEL** and the rules of instantiation, this result is equal to the original equation.²⁶

We now return to the discussion of **META** rules. As noted, a **META** rule may be generated from formulas of the form

```
(EQUAL (EVAL$ T v a)
        (EVAL$ T (simp v) a)),
```

where **simp** is a function symbol of one argument and **v** and **a** are distinct variable symbols.

When the user tags such a formula to be in the **META** class, a list of target function symbols, **fn₁**, ..., **fn_n**, must also be provided. The form of the tag must be

```
(META fn1 ... fnn)
```

where each **fn_i** is a function symbol. This is the only case in which the tag classifying the rule to be generated is not simply the Lisp atom naming the class. The system generates a **META** rule “**simp** is a metatheoretic simplifier for **fn_i**” for each **i**.

The selection of the **fn_i** is entirely up to the user and does not affect the validity of the rules generated. For example, had **CANCEL** been stored as a metatheoretic simplifier for **PLUS** or **PERM** it would not hurt anything but the efficiency of the system. The theorem establishing the correctness of the simplifier does so for all terms (indeed, even for non-terms). That is, the theorem proved about (**simp v**) is that its value is equal to that of **v**, for all **v**, not just for those with the “expected” function symbols. To define **simp** so that the correctness result can be proved in this form usually requires explicit inspection of the topmost function symbol of the argument term—even though the user “knows” that it will only be applied to terms beginning with an **fn_i**²⁷—and an explicit clause in the definition that returns the input term in the event that it was of an “unexpected” kind.

²⁶We would like to thank Matt Kaufmann for this generalization of our previous handling of metafunctions.

²⁷Such reasoning is invalid. Since nontame terms are generalized to tame ones before applying the metafunction it is possible for the metafunction to be applied to a variable symbol.

11.3. ELIM

An example **ELIM** rule is “**Eliminate the destructor terms** (**REMAINDER X Y**) and (**QUOTIENT X Y**) **by replacing x with the constructor term** (**PLUS (REMAINDER X Y) (TIMES Y (QUOTIENT X Y))**) and **generalizing, when** (**AND (NUMBERP X) (NOT (ZEROP Y))**).”

In general, an **ELIM** rule is a rule of the form “**Eliminate the destructor terms** d_1, \dots, d_n **by replacing x with the constructor term c-term and generalizing, when hyps,**” where the d_i , **c-term**, and **hyps** are terms, x is a variable, and the set of d_i s satisfy the “destructor term restriction” for **hyps**, **c-term**, and x . We call x the *elimination variable* of the rule.

A set of terms (henceforth called “destructor terms”) satisfies the *destructor term restriction* for x , **c-term**, and **hyps** iff (a) every destructor term is a “first-level function application” (see below), (b) no two destructor terms have the same outermost function symbol, (c) each variable occurring in x , **c-term**, or **hyps** occurs in each destructor term, (d) each destructor term occurs in **c-term**, and (e) x does not occur in **c-term** except inside the destructor terms.

We say a term is a *first-level function application* if it is of the form (**fn** $v_1 \dots v_n$), where **fn** is a function symbol and the v_i are all variables and all distinct.

The process of destructor elimination looks for subterms of the input formula that match with the destructor terms of some enabled **ELIM** rule, under some substitution σ that instantiates the elimination variable, x , of the rule with a variable symbol, v . If more than one set of function symbols can be eliminated, the process chooses the most “complicated” according to a heuristic. Once a rule and substitution have been chosen, the elimination proceeds in the two steps described earlier. First the elimination variable’s image, v , is replaced by the image of the constructor term, **c-term**/ σ , in all occurrences outside the destructor term images. Then the destructor term images are generalized to new variable symbols, taking account of type set information and the **GENERALIZE** rules available.

Unless further constrained, the destructor elimination process may loop indefinitely in concert with the simplifier. Heuristics are used to stop such loops by not eliminating on a variable introduced by elimination.

A formula can generate an **ELIM** rule only if the formula has the form

(**IMPLIES hyps (EQUAL c-term x)**)

where x is a variable symbol and the proper subterms of **c-term** which are first-level function applications, d_1, \dots, d_n , satisfy the destructor term restriction for x , **c-term**, and **hyps**. The **ELIM** rule generated is “**Eliminate the**

destructor terms d_1, \dots, d_n by replacing x with the constructor term **c-term** and generalizing, when hyps.”

On page 412 we give some guidelines on using **ELIM** rules.

11.4. GENERALIZE

Any term can be a **GENERALIZE** rule.

Suppose a term t is to be generalized by replacing it with a variable v in some formula. The enabled **GENERALIZE** rules are used to restrict v . This is done by adding instances of the **GENERALIZE** rules as hypotheses to the formula being generalized, before t is replaced by v . Instances are found by considering each rule, **gen**, and determining whether some nonvariable subterm of **gen** matches, under some substitution σ , the term t being generalized. If such a subterm and σ are found, **gen**/ σ is added as a hypothesis before t is replaced by v . After all such hypotheses are added, t is replaced by v in the modified formula.

12 Reference Guide

We now present an alphabetical listing of the commands to the theorem prover along with the definitions of concepts used in the command descriptions.

12.1. Aborting or Interrupting Commands

Means for aborting a computation or interrupting and resuming a computation are not specified in *Common Lisp The Language*, but there always seems to be an implementation dependent means for doing these things. Of course, even if it is possible to abort or interrupt a process, the exact point in the computation where the effect will occur is bound to be very time dependent, i.e., it's a random thing to do, in the vernacular.

In general, aborting a computation means that control is taken away from the ongoing computation and is given to the top-level read-eval-print loop of Common Lisp. To abort a computation:

- In Lucid on a Sun, simultaneously press the **CONTROL** and **c** keys and then type **:A** to the resulting break prompt.
- In GCL or CMU Common Lisp on a Sun, simultaneously press the **CONTROL** and **c** keys and then type **:Q** to the resulting break prompt.
- In Allegro on a Sun, simultaneously press the **CONTROL** and **c** keys and then type **:RES** to the resulting break prompt.

It is sometimes necessary to try several times to abort because certain Lisp programs (e.g., the garbage collector) may cause keystrokes to be ignored. In addition, if the theorem prover is printing to the terminal when you try to abort, you may get several pages of output before the requested action seems to occur. This is because terminal output lags behind the actual computation (e.g., because of buffering).

By “interrupting” a computation we mean that control is temporarily taken away from the ongoing computation and given to an inferior read-eval-print loop. From within this loop you can evaluate Lisp expressions, e.g., to inspect the state. See **BREAK-REWRITE** for example. When you are ready for the interrupted computation to resume, you communicate this to the inferior read-eval-print loop, which then terminates and returns control to the interrupted computation. To interrupt a computation

- In Lucid on a Sun, simultaneously press the **CONTROL** and **c** keys. To resume, type **:C** to the break prompt.
- In GCL on a Sun, simultaneously press the **CONTROL** and **c** keys. To resume, type **:R** to the break prompt.
- In Allegro on a Sun, simultaneously press the **CONTROL** and **c** keys. To resume, type **:CONT** to the break prompt.
- In CMU Common Lisp on a Sun, simultaneously press the **CONTROL** and **c** keys. To resume, type **:GO** to the break prompt.

Warning: It is technically dangerous to abort any command that changes the data base, including the event commands such as **ADD-SHELL**, **DEFN**, and **PROVE-LEMMA**. Such aborts may leave the data base in an inconsistent state. The only proofs we endorse are those constructed by an uninterrupted sequence of event commands.

Having thus officially disavowed the practice, let us now make a few practical remarks about the effects of aborting event commands. All commands that change the data base adhere to the policy of first storing “undo” information and then making the change. The undo information allows us later to remove an event from the data base. Thus, if you do abort an event command that might have made changes to the data base, use **UNDO-NAME** to remove the aborted event.

What do we mean by “that might have made changes to the data base?” How can you tell? The answer is, technically, see if the new name has an **EVENT-FORM** in the data base (see **DATA-BASE**). However, it doesn’t hurt anything to do an **UNDO-NAME**; the worst that will happen is that an error is reported because the “event” to be undone does not exist.

However, we personally behave in a somewhat less rigid way. Whenever we abort a **DEFN** or **ADD-SHELL** we do an **UNDO-NAME**. We don't bother to do an **UNDO-NAME** when we abort a **PROVE-LEMMA** because **PROVE-LEMMA** doesn't change the data base until after the proof has successfully completed. Because the theorem prover's output lags behind its actual computation, it is possible that we will someday abort a proof—thinking that it is doomed to fail—after it has in fact succeeded. But it has not happened yet. If it happens, the inconsistency will show up if the same event name is submitted later. In any case, we never feel like we have completed a proof project until an uninterrupted run of the events is performed by **PROVE-FILE** (see page 356).

12.2. ACCUMULATED-PERSISTENCE

General and Example Form:
(**ACCUMULATED-PERSISTENCE**)

This routine prints a summary of the persistence totals accumulated against function and rule names for the most recently started proof attempt during which the rewrite path was maintained. See **BREAK-LEMMA** and **MAINTAIN-REWRITE-PATH** for an explanation of related features.

The rewrite path is a stack of “frames” maintained by the rewriter and encodes a complete description of the currently active calls to the rewrite routine. Most frames record a call of the rewriter; each such frame is associated with a nonvariable term (the term being rewritten) or a rule (the rule being applied). When the rewriter has completed the processing of the term or rule, the frame is popped off the rewrite path. The number of frames built while a given frame is on the path is called the persistence of the frame and is an indication of the amount of work attributable to the term or rule associated with the frame. When we pop a frame, we compute its persistence and add it into an accumulating persistence table under the topmost function symbol of the term or the name of the rule. Thus, if a proof (successful or otherwise) has been run while the rewrite path was being maintained, the accumulated persistence table will indicate the “expense” of dealing with all of the function names and rules involved in the proof.

ACCUMULATED-PERSISTENCE prints to ***STANDARD-OUTPUT*** a summary of the accumulated persistence table. In particular, it says how many names have been seen by the rewriter and it lists the persistence totals of the 20 most persistent user-introduced names and the 5 most persistent primitive names. If names that are irrelevant to your proof appear in this listing, you could **DISABLE** them to speed up the proof. **ACCUMULATED-PERSISTENCE** actually takes two optional arguments, which default to 20 and 5.

12.3. ADD-AXIOM

General Form:

```
(ADD-AXIOM name rule-classes term)
```

Example Form:

```
(ADD-AXIOM NUMBERP-LOC (REWRITE) (NUMBERP (LOC X M)))
```

ADD-AXIOM is an event command for adding a new axiom to the logic and storing it and possibly some generated rules in the data base. It does not evaluate its arguments. **name** must be a new name and will be made the name of the event in the data base, **rule-classes** must be a (possibly empty) list of rule classes and **term** must be a well-formed term (see **TRANSLATE**). **ADD-AXIOM** stores **term** in the data base as an axiom; in addition, rules of the classes specified by **rule-classes** are generated from **term** and stored. Each rule has the name **name** and is initially **ENABLED**. An error is caused if **term** is unsuitable for some class in **rule-classes** and no change is made in the data base. If successful, **ADD-AXIOM** prints a time triple for the event and returns the name of the new event, **name**.

When formulating input for **ADD-AXIOM** you must be cognizant of the rule interpretation of **term** in addition to its mathematical content. See Chapter 11 for a detailed explanation of the effect of each type of rule and how rules are generated from formulas and rule classes. See Chapter 13 for advice.

Note that if **rule-classes** is **NIL** then the term is stored as an axiom and will be available for **USE** hints but will generate no rules.

We strongly advise against the use of **ADD-AXIOM**. Moderate use of the theorem prover usually provides convincing evidence for the proposition that users frequently believe formulas to be valid when they are not. Conferring upon an unproved formula the stamp **Axiom** in no way lessens the danger that it is inconsistent with the other axioms. If a concept or relation can be defined within the logic, we urge you to so define it. If a formula is in principle provable, we urge you to prove it. If your intention is to constrain some new function symbols to have certain properties, we urge you to use **CONSTRAIN**.

12.4. ADD-SHELL

General Form:

```
(ADD-SHELL const base r ac-descriptors)
```

Example Form:

```
(ADD-SHELL PUSH EMPTY-STACK STACKP
  ((TOP (ONE-OF NUMBERP LITATOM) ZERO)
   (POP (ONE-OF STACKP) EMPTY-STACK)))
```

ADD-SHELL is an event command for adding a new shell to the logic. It does not evaluate its arguments. **ac-descriptors** must be a list of **n** elements. Each element must be a triple of the form (**ac_i** **tr_i** **dv_i**). If **base** is non-**NIL**, the command has the theoretical effect of the axiomatic act

Shell Definition

Add the shell **const** of **n** arguments
with base function symbol **base**,
recognizer symbol **r**,
accessors **ac₁**, ..., **ac_n**,
type restrictions **tr₁**, ..., **tr_n**, and
default functions **dv₁**, ..., **dv_n**.

If **base** is **NIL** no base function symbol is supplied. Note that the **ADD-SHELL** command causes an error if the corresponding invocation of the shell principle is inadmissible in the current history. If admissible, the command adds a new event in the data base whose name is **const**. In addition, the command adds a large number of rules to the data base. The names of the rules are generated from the user-supplied names above. We do not describe the rules generated or their precise names here. The command prints a time triple and returns **const**.

It is with **ADD-SHELL** that **BOOT-STRAP** builds in the rules for the natural numbers, lists, and the other primitive shells. The only ways in which those primitive shells are “more built-in” than user declared shells are (a) because they are satellites of **GROUND-ZERO** they cannot be undone individually and we do not keep track of references to them, (b) the **QUOTE** notation provides succinct abbreviations for them, and (c) the linear arithmetic procedure is built specifically for the **NUMBERPs**.

Note. The time **ADD-SHELL** takes to operate depends on the number of accessors, **n**, and the number of recognizers, **k**, listed in the **ONE-OF** and **NONE-OF** clauses of the type restrictions. Roughly speaking, the size of the normal form of one of the rewrite rules added by **ADD-SHELL** grows exponentially with **n** and quadratically with **k**. The rewrite rule in question, called **const-EQUAL**, is essentially a conjunction of **n** terms, each of which is a disjunction of **k** terms. Computing the normal form of **const-EQUAL** can take a while, for large **n** and **k**. In the pre-1992 releases of Nqthm this problem was exacerbated by the coding of the routine for storing rewrite rules. That routine processed each rule in a way that was quadratic in the size of the (normalized) rule. As a result,

multi-accessor shell declarations with nontrivial type restrictions on each component could cause **ADD-SHELL** to take hours of cpu time. Nqthm-1992 contains a short cut, inspired by a suggestion of Matt Kaufmann's, that makes the storage processing linear for **const-EQUAL**. Nevertheless, the exponential growth is still inherent in the shell axioms. If you have an n-ary shell declaration that is taking a very long time, you might consider trivializing all of the type restrictions, by using **(NONE-OF)** for each accessor position. That will introduce a class of n-tuples with no restrictions on the components. You may then define a predicate that recognizes when such an n-tuple has the desired types of components. The **IF**-normal form of that predicate will suggest to you the size of the problem. If you use that predicate in your theorems, it may possibly cause combinatoric explosions. However, you can deal with them through conventional means: decompose your predicate into a nest of nonrecursively defined functions and disable those functions.

12.5. AXIOM

General Forms:

```
(AXIOM name rule-classes term)
and
(AXIOM name rule-classes term hints)
```

Example Form:

```
(AXIOM NUMBERP-LOC (REWRITE) (NUMBERP (LOC X M)))
```

The form **(AXIOM name rule-classes term)** is just an abbreviation for **(ADD-AXIOM name rule-classes term)**. The **hints** argument, if supplied, is ignored. This odd functionality is provided so that during proof development **AXIOM** commands can be used in place of selected **PROVE-LEMMA** commands—as one might want to do if an event list must be “replayed” to reproduce some data base state (e.g., to demonstrate a bug or rewrite-rule loop) but some **PROVE-LEMMA** commands in it take “too long.” See also **SKIM-FILE**.

12.6. BACKQUOTE-SETTING

General and Example Forms:

(BACKQUOTE-SETTING 'ORIG)

and

(BACKQUOTE-SETTING 'NQTHM)

If you do not use backquote, or at least, do not use backquote in the formulas you submit to Nqthm, then **BACKQUOTE-SETTING** is unimportant to you. If you do use backquote in your theorems, and especially if you use it both in Nqthm formulas and in special-purpose Lisp utilities you use in connection with Nqthm, then **BACKQUOTE-SETTING** is very important.

The Nqthm user interface is just the read-eval-print loop of the host Common Lisp. Commands are read with the standard Lisp reader. This raises a problem, illustrated below, because the backquote notation has different interpretations in different Common Lisps. Depending on which Common Lisp is used, an utterance involving backquote might read as a theorem, a non-theorem, or an ill-formed formula. To avoid this dependency on the host's interpretation of backquote, Nqthm supplies an interpretation that is consistent with the Common Lisp standard [131, 132]. If you use backquote in Nqthm terms, then you should be sure the terms are read under Nqthm's interpretation of backquote. However, the s-expressions produced by Nqthm's backquote, while consistent with the standard, are generally not as efficient, when treated as Lisp code, as those produced by most Common Lisps. Thus, if you use backquote in Lisp utility functions, you might wish to use the host interpretation when those utilities are loaded.

Common Lisp's "current readtable" (***READTABLE***) determines how the Common Lisp reader interprets backquote. The function **BACKQUOTE-SETTING** modifies the readtable so as to provide either of the two interpretations backquote.

(BACKQUOTE-SETTING 'ORIG) modifies the current Common Lisp readtable so that backquote is given the interpretation initially present in the host Common Lisp. That is, after invoking **(BACKQUOTE-SETTING 'ORIG)**, type-in is read according to the host's interpretation of backquote. **(BACKQUOTE-SETTING 'NQTHM)** modifies the current Common Lisp readtable so that backquote is given the interpretation defined in Section 4.7.

Warning. The initial interpretation of backquote in Nqthm is that of the host Common Lisp. If you want our formal interpretation in a session, you must invoke **(BACKQUOTE-SETTING 'NQTHM)**.

We now illustrate the problem caused by backquote. The Common Lisp standard [131, 132] does not specify the s-expression read in response to a back-

quote expression. Instead, it specifies what the value of that s-expression must be. Different implementations of Common Lisp produce different s-expressions for the same backquote expression. For example, the Nqthm interpretation of ``(,X)` is `(CONS X 'NIL)`; the GCL (Gnu Common Lisp) interpretation of the same backquote expression is `(LIST X)`; the Lucid interpretation is `(LUCID-RUNTIME-SUPPORT:BQ-LIST X)`. Thus, `(EQUAL (CAR '(,X)) 'CONS)` reads as a theorem under Nqthm's interpretation of backquote, reads as a falsehood under GCL's, and reads as an ill-formed formula under Lucid's because it involves the character sequence `LUCID-RUNTIME-SUPPORT:BQ-LIST` which is not a legal word (see page 143) because it contains a colon.

Many experienced Nqthm users have special-purpose Lisp programs and macros they use in connection with Nqthm. Such users must be especially careful about which interpretation is given backquote. When their Lisp utilities are read and compiled, the **ORIG** backquote setting should be in use, for efficiency. When Nqthm formulas are read, the **NQTHM** setting should be in use.

The Nqthm utility **PROVE-FILE**, which reads Nqthm events from a file and executes them, always uses the Nqthm interpretation of backquote, regardless of what setting is in force in Common Lisp's read-eval-print loop. This is part of **PROVE-FILE**'s assurance that the file contains nothing but well-formed events. **PROVE-FILE** restores the previous interpretation upon termination.

Our decision to make the host's backquote setting be the default setting stems from the fact that few Nqthm users are tempted to use backquote in Nqthm terms. But this decision admits the following unhappy scenario: The user fires up a Lisp and begins to develop a sequence of events leading to some interesting result, submitting each event to the read-eval-print loop as development proceeds. At some point, backquote is used in an event. The user, however, has forgotten the backquote problem and is accidentally still using the host's interpretation. It happens that the host in question expands backquoted forms into legal Nqthm terms but different ones than Nqthm's interpretation would produce. Thus, without knowing it, the user develops a data base of rules around this spurious interpretation of type-in. When the final event list is submitted to **PROVE-FILE**, which uses the correct Nqthm interpretation of backquote, it fails to replay. The moral is that if you use backquote in terms, use **BACKQUOTE-SETTING** to ensure you get the correct treatment of your type-in. Unfortunately, it is sometimes difficult, in the heat of the chase, to note that you've begun to use backquote.

12.7. BOOT-STRAP

General Form:

(BOOT-STRAP flg)

Example Form:

(BOOT-STRAP NQTHM)

BOOT-STRAP is an event command. It erases the current data base and initializes it to the Ground Zero logic. This command creates a very large number of rules corresponding to the primitive shells and the definitions of the functions in the Ground Zero logic. It prints a time triple and returns the event name **GROUND-ZERO**.²⁸

The **flg** argument lets the user select which of two sets of Ground Zero axioms should be used. The argument is not evaluated by Lisp. If **flg** is **NQTHM** the theory used is that documented in this handbook. If no **flg** argument is supplied or if **NIL** is supplied or if **flg** is **THM**, the system is booted into a constructive logic very similar to the one we used before adopting the current one. We call the first logic “**NQTHM** logic” (for “New Quantified THM”) and say the theorem prover is running in “**NQTHM** mode” when that logic is being used. We call the second logic “**THM** logic” and the associated mode “**THM** mode.”

THM logic differs from **NQTHM** logic in the following respects:

- **THM** does not support the ordinals. **ORD-LESSP** and **ORDINALP** are not defined in **THM**. Two simple lexicographic relations, **LEX2** and **LEX3**, are defined. **(LEX2 (LIST i₁ j₁) (LIST i₂ j₂))** is **T** precisely if either **(LESSP i₁ i₂)** or **(EQUAL i₁ i₂)** and **(LESSP j₁ j₂)**. **LEX3** is defined similarly. **LESSP**, **LEX2**, and **LEX3** are the only accepted well-founded relations in **THM**.
- **THM** does not support **V&C\$** but does provide a more restricted sense of **EVAL\$**. In particular, **ASSOC**, **PAIRLIST**, **FIX-COST**, **STRIP-CARS**, **SUM-CDRS**, **SUBRP**, **APPLY-SUBR**, **FORMALS**, **BODY**, **V&C\$**, and **V&C-APPLY\$** are defined in **NQTHM** but are not defined in **THM**. **EVAL\$** and **APPLY\$** are axiomatized in **THM** but they are not equivalent to their counterparts in **NQTHM**. **(EVAL\$ T X A)** in **THM** is equivalent to **(MEANING X A)**, as it is defined in [25]. **(EVAL\$ 'LIST X A)** is equivalent to **(MEANING-LST**

²⁸A quick and dirty sketch of the theory created by **BOOT-STRAP** is available by printing the value of the Lisp variable **BOOT-STRAP-INSTRS**.

X A) of [25]. **APPLY\$** in **THM** is axiomatized as is **APPLY** in [25]. Roughly speaking this means that **EVAL\$** “does not know its own name.” Additional restrictions are enforced in **THM**: no definition body may use **EVAL\$** or **APPLY\$**. The consequence of these restrictions is that mapping functions like **FOR** cannot be defined, but **EVAL\$** of **'term** is always equal to **term** under the standard alist, provided **term** does not mention **EVAL\$** or **APPLY\$**. The only intended use for **EVAL\$** in **THM** is in the statement of correctness for metafunctions.

- **THM** does not support **FOR**. Hence, the functions included in the **NQTHM** Ground Zero theory exclusively for the definition of **FOR** are not defined in **THM**: **ADD-TO-SET**, **APPEND**, **MAX**, **UNION**, **QUANTIFIER-INITIAL-VALUE**, **QUANTIFIER-OPERATION**, and **FOR**.
- **THM** contains definitions for the functions **LENGTH**, **SUBSETP**, and **LAST**, which are not defined in the Ground Zero logic of **NQTHM**.

THM logic differs somewhat from the logic supported by the previous release of our theorem prover. Let us call the previous version of the logic the “old” logic. The differences between **THM** logic and the old logic are as follows:

- **IFF** is included in **THM** but not in the old logic. This permits the implementation, in **THM** mode, of “propositional replacement” rules. If you have an old-style event list including a definition of **IFF**, you should delete it (if it is defined as is our **IFF**) or rename it and all its uses otherwise.
- **EVAL\$** and **APPLY\$** are included in **THM** but not in the old logic. In fact, they “replace” the functions **MEANING**, **MEANING-LST**, and **APPLY** of the old logic. The implementation of metalemmas in **THM** and **NQTHM** was simplified by changing the names of **MEANING** and **APPLY** to **EVAL\$** and **APPLY\$**. See the next note. If you have an old-style event list involving theorems about **MEANING** and **APPLY**, use **EVAL\$** and **APPLY\$** instead.
- The form of metalemmas in **THM** is as described here, not as described in [25]. The old style of metalemmas required proving that the metafunction returns a “**FORMP**”. The new style does not. If you have an old-style event list containing a metalemma, reformulate it as described here. The new theorem will be simpler.
- In addition to **MEANING**, **MEANING-LST**, and **APPLY**, the following functions were defined in the old logic but are not defined in **THM**: **FORMP**, **FORM-LSTP**, **ARITY**, **SYMBOLP**, **LEGAL-CHAR-CODE-SEQ**, **ILLEGAL-FIRST-CHAR-CODES**, and **LEGAL-CHAR-CODES**. These functions all participated in the

definition of **FORMP** and hence were necessary for the old-style form of metalemma. Because they are no longer necessary for metalemmas, and because we imagine that no user relied upon them except for metalemmas, we simply omitted putting them into the **THM** logic.

12.8. BREAK-LEMMA

General Form:

(BREAK-LEMMA name)

Example Form:

(BREAK-LEMMA 'REVERSE-REVERSE)

The argument, **name**, is evaluated and should be the name of a replacement rule or linear arithmetic rule. **BREAK-LEMMA** alerts the rewriter to look out for all attempted applications of the named rule. Whenever the rule is tried, an interactive break occurs. To remove **name** from the list of monitored rules, use **UNBREAK-LEMMA**. Note that **BREAK-LEMMA** cannot be used to break on function definitions or on type-prescription, compound recognizer, meta, elimination, or generalization rules. If **name** is not the name of an **ADD-AXIOM**, **CONSTRAIN**, **FUNCTIONALLY-INSTANTIATE** or **PROVE-LEMMA** event with lemma type **REWRITE**, **BREAK-LEMMA** prints a warning message but alerts the rewriter anyway. It is most often the case that this warning message means you misspelled the name of the rule to be monitored, but it is sometimes useful to break on names that are satellites of other events (e.g., a rewrite rule introduced by an **ADD-SHELL**) so the simple test that **name** names an event is insufficient.

Warning. Because it is possible to execute an arbitrary Common Lisp form while in the interactive break under the theorem prover, it is possible to do arbitrary damage. For example, executing **(THROW 'PROVE T)** will immediately terminate the proof attempt successfully! We do not endorse proofs in which interactive breaks occur. We provide **BREAK-LEMMA** and **BREAK-REWRITE** simply as a means to help you diagnose what is going wrong with your rewrite rules.

Recall that to apply a replacement or linear arithmetic rule the first step is to find a substitution under which the left-hand side of the replacement rule or the key term of the linear arithmetic rule matches some target term being simplified. The substitution is computed by a simple pattern matcher that is complete: if a substitution exists to make the rule syntactically identical to the target, we find it (quickly). Subsequent steps in the application of a rule include relieving the

hypotheses and rewriting the right-hand side of the conclusion. (See pages 279 and 286.)

When a matched rule's name appears on the list of monitored rules, an interactive break occurs immediately after the pattern matcher has succeeded. When the pattern matcher fails, no substitution exists and no break occurs even though the rule was actually considered briefly. Thus, if a rule is being monitored and no break for it occurs during a proof attempt, then either the rule is disabled or no instance of its left-hand side (or key term, as the case may be) was ever encountered by the rewriter. In this case it is probable that your expected target term got mangled somehow by other rules. We offer no mechanical help for tracking down the rules or identifying the mangled target. Our best advice is to think hard about what rules might have applied to your target. Some users would here use *Pc-Nqthm* [75, 21] to carry out the rewrites which supposedly produce the target term, to verify that the term is as expected; *Pc-Nqthm* can also be used to determine all the rules applicable to a given term, which might suggest how the intended term got mangled.

If a substitution is found, then an interactive break occurs. By typing various commands you can inspect the state of the rewriter and step through the process of attempting to apply the rule. For replacement rules the steps are first relieve the hypotheses and then rewrite the right-hand side of the conclusion. For linear arithmetic rules there are more steps: relieve the hypotheses, rewrite the conclusion, convert the rewritten conclusion into polynomial form, and make heuristic checks on the resulting polynomials. Keep in mind that additional breaks may occur during the recursive processing done to relieve hypotheses and rewrite conclusions.

If at any step it is determined that the rule cannot be applied, the interactive break provides commands for displaying an explanation. However, the interactive break is not intended as a proof-checker: no facilities are provided for directing the theorem prover's strategy or affecting in any way the outcome of the attempt to apply the rule.

The interactive break is a Lisp read-eval-print loop where certain atoms are treated specially. The name of the Lisp function that implements the command interpreter is **BREAK-REWRITE** which, under certain conditions, can be invoked directly by the user during Lisp breaks to inspect the state of the rewriter (see page 312). The prompt character for the break is a colon. Your commands are read by the Lisp reader. The command ? (i.e., the Lisp symbol obtained by reading a question mark followed by a carriage return) will cause a brief summary of the commands to be printed.

The available commands are divided into two categories: those that are specific to this particular step in the attempt to apply the rule and general commands for inspecting the state of the rewriter. The general commands are available all the time. The context-specific or "special" commands change

with each step. For example, on entry to the break, the command **OK** means “try to relieve the hypotheses and break again when the attempt has finished.” But immediately after the hypotheses have been relieved, **OK** means “proceed to rewrite the conclusion and break again when that has finished.” In addition, the special commands at any particular time may include some that make sense only at that time. For example, one can ask to see the **TARGET** term anytime, but one can ask for the **FAILED-HYP** only after a failed attempt to relieve the hypotheses.

Roughly speaking, the special command **OK** always means “go ahead with the next step and break when it is done” while the special command **GO** always means “go ahead with the next step and all subsequent steps of this application.” **GO** is useful when you wish to shortcut the interactive processing of a rule because you have determined (say from the **TARGET** term) that the application at hand is not interesting.

Until you are familiar with the special commands you should type “?” often to see what your options are.

The general commands for inspecting the state of the rewriter are not summarized by the “?” command, since “?” is typed fairly often. A summary of the general commands may be obtained by typing “??”. Using the general commands you can inspect the “rewrite path” from the current target term up to the top-level goal of the simplifier. The rewrite path is a stack of frames. Roughly speaking, each frame (except the first) represents a call of the theorem prover’s rewrite routine. Commands permit you to see the entire list of frames sketchily or to focus attention on a particular frame and get more information. The frame on which attention is focused is called the “current frame” and is initially the frame in which the target term is being rewritten. You may use general purpose commands to move the current frame up or down the stack. Other commands display detailed information about the current frame. However, the position of the current frame in no way affects the simplifier or rewriter! That is, if you move the current frame up to a point where some other target is being rewritten, the theorem prover’s attention does not shift to that target! The notion of the “current frame” is merely a convenient way to let you inspect the stack.

We now present an example of the use of general commands. Suppose we have the rules generated from the following two events:

```
(PROVE-LEMMA REVERSE-REVERSE (REWRITE)
  (IMPLIES (PROPERP X)
    (EQUAL (REVERSE (REVERSE X)) X)))
```

```
(PROVE-LEMMA PROPERP-APPEND (REWRITE)
  (IMPLIES (PROPERP B) (PROPERP (APPEND A B))))
```

Suppose also we have installed a break on **PROPERP-APPEND**:

```
(BREAK-LEMMA 'PROPERP-APPEND)
```

Finally, suppose we then execute the event

```
(PROVE-LEMMA LEMMA NIL
  (IMPLIES (AND (PROPERP B)
                (PROPERP A))
    (EQUAL
      (REVERSE (REVERSE (APPEND A B)))
      (APPEND A B))))
```

We show the resulting output below in **typewriter** font.

```
(Entering break on replacement rule PROPERP-APPEND
```

```
: ?
```

You are in the **BREAK-REWRITE** command interpreter.
The commands specific to this break are:

cmd	effect
OK	Attempt to relieve the hypotheses and then break.
GO	Proceed without further interaction.
NAME	Print name of the broken rule.
HYPs	List the hypotheses of the broken rule.
CONCL	Print the conclusion of the broken rule.
SUBST	Print the substitution being applied to the broken rule.
TARGET	Print the term to which the rule is being applied.
??	General purpose commands.

```
: TARGET
```

```
(PROPERP (APPEND A B))
```

Note that we are trying to use **PROPERP-APPEND** to prove that **(APPEND A B)** is **PROPERP**. We might wish to know how this subgoal has arisen. To find out, we use the general purpose commands. First we use **??** to get a summary of them.

```
: ??
```

You are in the BREAK-REWRITE command interpreter.
The general purpose commands are:

cmd	effect
PATH	Highlight the REWRITE path.
FRAME	Print the current frame, pruning deep terms.
PATH!	Print every frame in the path, pruning deep terms.
FRAME!	Print the current frame, with no pruning.
PATH!!	Print the path, with no pruning.
ASSUMPTIONS	Print the governing assumptions.
ANCESTORS	Print the negations of backchaining hypotheses.
ACCUMULATED-PERSISTENCE	Print the accumulated persistence totals.
BK	Move one frame towards the top-level SIMPLIFY.
NX	Move one frame away from the top-level SIMPLIFY.
TOP	Go to the top-level SIMPLIFY frame.
BTM	Go to the frame farthest from the top-level SIMPLIFY.
n (a natural number)	Go to frame number n.
s-expr (a list expression)	Evaluate Common Lisp s-expr
?	Special purpose commands.

The rewrite path will tell us how we got to the current target from the top-level goal of the simplifier. The sketchiest display of the rewrite path is given by the **PATH** command:

```
: PATH
( 90) 0. (top)
( 62) 1. Rewriting (EQUAL ...)
*( 61) 2. Rewriting (REVERSE ...)
( 3) 3. Applying REVERSE-REVERSE
*( 2) 4. Rewriting (PROPERP ...)
( 0) 5. Applying PROPERP-APPEND
```

There are six frames on the stack, numbered 0 through 5. The topmost one, 0, is the initial call of the simplifier. The bottommost one, 5, is the frame in which we are working on the target term. From the above display we can see that we got to the target from the top by (a) rewriting an **EQUAL** expression in

the topmost goal, (b) rewriting a **REVERSE** expression below that, (c) attempting to apply the rule **REVERSE-REVERSE** to that, (d) rewriting a **PROPERP** expression as part of the attempt, and (e) attempting to apply **PROPERP-APPEND** to that. However, in this sketch of the path we are not told the relation of one frame to the next.

Note also the parenthesized numbers on the left-hand edge of the display. Each number is the “persistence” of the corresponding frame. The persistence of a frame is the number of frames built since the given frame was built. The persistence of a frame is an indication of the amount of work expended so far on behalf of that frame. A frame is marked with a “*” if the persistence of the frame is at least twice the persistence of the frame immediately below it. In some sense, the frames marked with *s contain terms or rewrite rules that have caused the rewriter a lot of work relative to the total amount of work so far.

From the fact that the persistence of frame 1 is 62 we know that 62 frames have been built since we began rewriting the **EQUAL** expression in frame 1. Since only four of them are now active (frames 2-5), we know the others have since been discarded. At first sight then, the **EQUAL** expression in frame 1 is relatively expensive to rewrite. But the persistence of frame 2 is 61. Thus, except for the construction of frame 2 itself, all the work involved in frame 1 has been done under frame 2. Now note the persistence of frame 3. It is only 3. Thus, 58 frames (61 minus 3) were built under frame 2 before we got around to trying **REVERSE-REVERSE** in frame 3. We conclude that the **REVERSE** expression in frame 2 was relatively expensive to rewrite and frame 2 is marked with a *. What work was done? After the frame was built we rewrote the arguments to the **REVERSE** expression before looking for rules that match the target. Thus, the 58 frames in question were used for the arguments and for any **REVERSE** rules tried before **REVERSE-REVERSE**. See also **ACCUMULATED-PERSISTENCE**.

The entire rewrite path is displayed in more detail by the following command:

```
: PATH!
---- Frame 0 ---- (persistence 90)
Goal:
(IMPLIES (AND (PROPERP B) (PROPERP A))
  (EQUAL (REVERSE #) (APPEND A B)))

---- Frame 1 ---- (persistence 62)
Rewriting the conclusion of the top-level goal:
(EQUAL (REVERSE (REVERSE #))
  (APPEND A B))
under the substitution:
NIL
```

```

---- Frame 2 -----                (persistence 61)  <--***
Rewriting the first argument of the term in frame 1:
(REVERSE (REVERSE (APPEND A B)))
under the substitution:
NIL

```

```

---- Frame 3 -----                (persistence 3)
Attempting to apply the replacement rule
REVERSE-REVERSE using the substitution:
X <- (APPEND A B)

```

```

---- Frame 4 -----                (persistence 2)  <--***
Rewriting the first hypothesis of REVERSE-REVERSE:
(PROPERP X)
under the substitution:
X <- (APPEND A B)

```

```

---- Frame 5 -----                (persistence 0)
Attempting to apply the replacement rule
PROPERP-APPEND using the substitution:
B <- B
A <- A

```

Observe that now we see more of each term being rewritten. However, we do not see each term in its entirety—note frame 1. The symbol # is printed in place of deep subterms. This is what is meant by “pruning” in the ?? command summary. **PATH!** also gives an explanation relating the activity in each frame to the activity in the frame above. In particular, we see that the **REVERSE** expression being rewritten in frame 2 is in fact **(REVERSE (REVERSE (APPEND A B)))** and is the first argument of the **EQUAL** expression being worked on in frame 1.

When the stack contains hundreds of frames, **PATH!** often provides too much information. That is why the notion of current frame exists. The commands **FRAME** and **FRAME!** print out the current frame. The command **n**, where **n** is a natural number, means “select frame **n** as the current frame.” The commands **BK** (for “back”) and **NX** (for “next”) let you move the current frame back and forth and are useful for exploring an area of the path.

12.9. BREAK-REWRITE

General Form:

(**BREAK-REWRITE**)

This Lisp routine permits the user to inspect the rewrite path (see page 307) as though a **BREAK-LEMMA** rule monitor had caused an interactive break. **BREAK-REWRITE** is a user-level entry into the command interpreter and can be invoked directly by the user after forcing a Lisp break while the rewriter is operating.

The method used to force a Lisp break while a computation is in progress is implementation dependent. See page 296. Once a break has been caused, executing (**BREAK-REWRITE**) will enable you to inspect the rewrite path with the general purpose commands described in **BREAK-LEMMA** above. (The special purpose commands (e.g., **TARGET** or **CONCL**) do not make sense since there is in fact no rule associated with the break.) The command **OK** exits **BREAK-REWRITE** and returns to the Lisp break from which it was invoked.

For efficiency, the theorem prover does not maintain the rewrite path all the time. If **BREAK-REWRITE** is called when the path is not being maintained, or when the rewrite path is empty (so that there is no topmost frame), it prints a message and returns immediately. See **MAINTAIN-REWRITE-PATH**.

BREAK-REWRITE is useful when you are trying to find out why the theorem prover is spending an unusually long time silently simplifying a particular formula. (Of course, what is unusual depends on what is usual for you. There are successful proofs in which the system is silent for many minutes.) If you have previously enabled the maintenance of the rewrite path you might force a Lisp break. If you have not previously enabled the maintenance of the rewrite path you should abort the now silent proof attempt, use (**MAINTAIN-REWRITE-PATH T**) to turn on maintenance, restart the proof attempt and force a break as before when the silent simplification is in progress. Once you have forced a break while the simplifier is running, call **BREAK-REWRITE** and use the path and frame commands to see what is going on. Most likely you will be surprised at the depth of the path and perhaps at the names of some of the rules or functions on it. Typically in these situations the simplifier is backchaining through rules you have long since forgotten and has set itself subgoals that have little to do with the problem at hand; another common problem is that nonrecursive functions have been expanded and have introduced many cases or concepts needlessly. In any case, it is frequently the case that by inspecting the path you will find the names of rules and functions that can be **DISABLED** to make the proof go faster.

In addition, **BREAK-REWRITE** is often useful in tracking down circularities

in your rewrite rules. Circularities in the rewrite rules lead to stack overflows in the rewriter. Often when stack overflow occurs, there is no way to investigate it, because the Lisp stack is already as big as permitted. However, the rewrite path maintained by the rewriter is not part of the stack but is a separate data structure that persists even after an aborted computation. If the rewrite path was being maintained during a proof attempt that caused a stack overflow, you may abort out of the stack overflow error—thereby gaining all that stack space. Then, from the top-level Lisp read-eval-print loop invoke (**BREAK-REWRITE**). If it reports that the theorem prover is not in the simplifier, it means “the theorem prover was not in the simplifier when the stack overflow occurred.”²⁹ Otherwise, you will be able to use the **BREAK-REWRITE** commands to investigate the rewrite path as it stood at the time of the stack overflow. This will often make the loop manifest.

This conjecture can be simplified, using the abbreviations **ADDRP** and **IMPLIES**, to:

```
Error: Bind stack overflow.
Error signaled by PROVE-LEMMA-FN.
```

```
Broken at PROVE-LEMMA-FN.  Type :H for Help.
>>:Q
```

```
Top level.
```

```
>(BREAK-REWRITE)
: PATH

(3404)  0. (top)
(3394)  1. Rewriting (GET ...)
(3392)  2. Applying  GET-NORMALIZER
(3382)  3. Rewriting (GET ...)
(3359)  4. Applying  GET-NORMALIZER
...
( 92) 202. Applying  GET-NORMALIZER
( 82) 203. Rewriting (GET ...)
( 59) 204. Applying  GET-NORMALIZER
( 49) 205. Rewriting (GET ...)
( 26) 206. Applying  GET-NORMALIZER
```

²⁹Some replacement rules are considered so simple that they are used as “abbreviations” and applied exhaustively *before* we begin normal simplification. These rules may contain loops, and at present we offer no aids in tracking down such loops.

```

( 16) 207. Rewriting (GET ...)
( 15) 208. Rewriting (ADD-ADDR ...)
( 13) 209. Rewriting (CONS ...)
*(  8) 210. Rewriting (PLUS ...)
*(  2) 211. Rewriting (IF ...)
(  0) 212. Rewriting (IF ...)

```

In using **BREAK-REWRITE** to look for circularities you must keep in mind that the path does not show everything that *has been used* to derive the current term, but what is being used right now. It might be that, in some sense, the “cause” of the loop is a rule not shown on the current path but which fires and completes, introducing a term which will permit it to fire again later during the rewriting.

12.10. CH

General Forms:

```

(CH)
and
(CH n)
and
(CH m n)

```

Example Forms:

```

(CH 10)
(CH 'PROPERP-REVERSE)
(CH 12 20)

```

CH prints out the names of the events that have been created so far. **CH** takes zero, one or two arguments which are used to specify an interval in the chronological sequence of events. The arguments are both evaluated and should be either event names or event numbers, where 0 is the most recent event, 1 the next most recent, etc. Events are printed in reverse chronological order, i.e., the youngest (most recently created) event is displayed first. If no arguments are supplied, the entire chronology is printed. If only one argument is supplied, it is taken as the rightmost end point, and 0 (the most recently created event) is used as the leftmost.

Here is an example. Suppose that we have just completed the proof described in the sample session starting on page 231. In particular, starting from **BOOT-STRAP** we defined **REVERSE** and **PROPERP** and then worked our way through the proofs of several lemmas about **PROPERP**, **APPEND**, and

REVERSE, concluding with **REVERSE-REVERSE**. Then the following use of **CH** prints out a sketch of the entire session in reverse chronological order:

```
(CH 'GROUND-ZERO)

0  (PROVE-LEMMA REVERSE-REVERSE ...)
1  (PROVE-LEMMA REVERSE-APPEND ...)
2  (PROVE-LEMMA APPEND-IS-ASSOCIATIVE ...)
3  (PROVE-LEMMA PROPERP-REVERSE ...)
4  (PROVE-LEMMA PROPERP-APPEND ...)
5  (PROVE-LEMMA APPEND-RIGHT-ID ...)
6  (DEFN PROPERP ...)
7  (DEFN REVERSE ...)
8  (BOOT-STRAP NQTHM ...)
```

Equivalent **CH** commands are **(CH 8)**, **(CH 0 8)**, and **(CH 'REVERSE-REVERSE 'GROUND-ZERO)**.

CH prints a **D** after the number if the event is currently disabled. Thus, suppose we execute **(DISABLE PROPERP-REVERSE)** and **(DISABLE PROPERP-APPEND)**. If we print the events between **REVERSE-REVERSE** and **APPEND-RIGHT-ID** we get

```
2  (PROVE-LEMMA REVERSE-REVERSE ...)
3  (PROVE-LEMMA REVERSE-APPEND ...)
4  (PROVE-LEMMA APPEND-IS-ASSOCIATIVE ...)
5 D (PROVE-LEMMA PROPERP-REVERSE ...)
6 D (PROVE-LEMMA PROPERP-APPEND ...)
7  (PROVE-LEMMA APPEND-RIGHT-ID ...)
```

Observe in the example above that the event numbers of the events shown have been increased by 2 because the two **DISABLE**s (actually **TOGGLE** events) have become events 0 and 1. In general, event numbers are so volatile that they are useful only in successive calls of **CH** as one scans through the chronology looking for a given interval. The only other command in our system that uses event numbers is **UBT**.

When the theorem prover is being used with a fast terminal and text editor it is often practical to print the entire chronology, as with **(CH)** and then scan it at leisure with the editor. But with slow terminals or systems that discard text that has been scrolled off the top, the primitive interval handling of **CH** is useful.

12.11. CHRONOLOGY

General Form:

CHRONOLOGY

The Lisp variable **CHRONOLOGY** is set to the list of all event names in reverse chronological order. Thus, the first element of **CHRONOLOGY** is the name of the most recently processed event. The last element is always **GROUND-ZERO**, the name of the oldest event. Users typically print the value of **CHRONOLOGY** to refresh their memory of “where they are” or what names they have used. Printing **CHRONOLOGY** is a good way to see what is in a data base that has just been loaded. However, the value of **CHRONOLOGY** is often very long and the command **CH** is often more convenient.

Users should not bind or set **CHRONOLOGY**.

12.12. COMMENT

General Form:

(COMMENT x_1 ... x_k)

Example Form:

(COMMENT THE NEXT EVENT IS THE CRUCIAL LEMMA)

COMMENT is defined as a Lisp macro that takes an arbitrary number of arguments and returns **T**. **COMMENT** ignores its arguments.

COMMENT is provided for historical reasons: an early version of Nqthm was written in Interlisp [137] and the **COMMENT** macro was the Interlisp primitive supporting the insertion of comments into code. The arguments to **COMMENT** were generally symbols or strings expressing some comment in “natural language” (e.g., English, German). Because of this Interlisp ancestry, users grew accustomed to inserting such comments in event lists, where the **COMMENT** form appeared as though it were an event command. **COMMENT** is not an event; it does not change **CHRONOLOGY** or add any names or rules. But we permit **COMMENT** forms in places where events forms are expected, e.g., **PROVE-FILE**.

Because Nqthm is now written in Common Lisp, we generally use the Common Lisp semicolon and number sign comments. Such comments have the advantage that they can be written virtually anywhere a “white space” can be written, e.g., before or after an event, or “inside” an event or formula, and they do not have to follow any syntactic rules except those regarding their delimiters.

12.13. COMPILE-UNCOMPILED-DEFNS

General Form:

```
(COMPILE-UNCOMPILED-DEFNS filename)
```

Example Form:

```
(COMPILE-UNCOMPILED-DEFNS "/temp/foo")
```

The argument, **filename**, is evaluated and should be a file name. This routine creates a file of the specified name, containing the Common Lisp definitions of the currently uncompiled executable counterparts (see page 337) of all functions in the current data base; it then compiles that file, and loads it. Thus, after calling this routine all executable counterparts are defined with compiled code. This will significantly increase the speed with which **R-LOOP** computes the values of submitted expressions and may increase the speed of the theorem prover if your proofs involve computation with explicit values.

The name of the file created has as its extension the value of the variable **FILE-EXTENSION-LISP**. The directory on which the file resides is affected by the value of the variable ***DEFAULT-NQTHM-PATH***. The extension of the file loaded after the compilation is specified by the variable **FILE-EXTENSION-BIN** (see page 339).

We do not always automatically compile each executable counterpart as it is created because in some circumstances (e.g., when using Kyoto Common Lisp) invoking the function **COMPILE** involves significant overhead and the creation of weirdly named temporary files. In addition, because files are created there is the possibilities of protection violations and resource allocation errors, problems that the theorem prover is not able to gracefully report much less recover from. However, if you wish to have each executable counterpart **COMPILED** as it is defined, you may set the flag ***COMPILE-FUNCTIONS-FLG*** to **T**. We advise against doing this except on Lisp machines.

12.14. CONSTRAIN

General Forms:

```
(CONSTRAIN name rule-classes term fs)
```

and

```
(CONSTRAIN name rule-classes term fs hints)
```


Example Form:

```
(CONSTRAIN H-COMMUTATIVITY2 (REWRITE)
  (EQUAL (H X (H Y Z))
    (H Y (H X Z)))
  ((H PLUS))
  ;Hints
  ((USE (PLUS-COMMUTATIVITY2))))
```

CONSTRAIN is an event command that introduces new function symbols that are constrained to satisfy a given axiom. It does not evaluate its arguments. **name** must be a new name and will be made the name of the event in the data base and **rule-classes** must be a (possibly empty) list of rule classes. **fs** must be a functional substitution; the domain of **fs**, $\{f_1 \dots f_n\}$, must consist entirely of new names and may not include **name**, and no **LAMBDA** expression in the range of **fs** may contain free variables. The s-expression **term** must be a term, *provided* that the symbols in the domain of **fs** are considered function symbols with the arities of their images under **fs**. Finally, **hints**, if provided, must be as required by the **PROVE-LEMMA** event (see page 364). The **CONSTRAIN** command has the same theoretical effect as the axiomatic act

Constraint.

Constrain f_1 , ..., and f_n to have property **term**,

with the additional restrictions that **fs** is a functional substitution satisfying the requirements of the axiomatic act and that **term**/**fs** is not just provable in the logic but provable by the implemented theorem prover.

Operationally, **CONSTRAIN** invokes the theorem prover on the functional instantiation of **term** under **fs**, i.e., on **term**/**fs**, supplying the prover with any **hints** provided. Note that **term**/**fs** is a term and if it is a theorem then there exist definitions of the functions in the domain of **fs** such that **term** is a theorem. Thus, if the proof succeeds, **CONSTRAIN** declares each of the symbols in the domain of **fs** to be a function symbol of the appropriate arity, adds **term** as an axiom, and in addition, generates from **term** the rules specified by **rule-classes** and stores each of them in the data base. The name of each rule is **name**. An error is caused and there is no change in the data base if **term**/**fs** cannot be proved or if **term** is of a form inappropriate for some class in **rule-classes**.

Because the only axiom added by **CONSTRAIN** is **term**, the values of the functions **BODY**, **FORMALS**, **SUBRP**, and **APPLY-SUBR** on the new function symbols are undefined.

When formulating input for **CONSTRAIN** you must be cognizant of the rule interpretation of **term** in addition to its mathematical content. See Chapter 11

for a detailed explanation of the effect of each type of rule and how rules are generated from formulas and rule classes. See Chapter 13 for advice about how to use rules.

Note that if **rule-classes** is **NIL** then the term is stored as a theorem and will be available for **USE** hints (see **PROVE-LEMMA**) but will generate no rules.

12.15. DATA-BASE

General Form:

(**DATA-BASE** *query name*)

Example Forms:

(**DATA-BASE** 'IMMEDIATE-DEPENDENTS' **REVERSE**)

and

(**DATA-BASE** 'IMMEDIATE-SUPPORTERS' **REVERSE**)

and

(**DATA-BASE** 'EVENT-FORM' **REVERSE**)

This is a low-level function for getting information about the data base. Before discussing the function we describe the data base in detail.

The data base is an acyclic directed graph. At each node is the following information:

- the name of the node,
- the event command which created the node,
- a number indicating the time at which the node was created (used to order event lists chronologically), and
- a list of satellites—the names of auxiliary axioms, functions, and executable counterparts introduced by the event command that created the node (see the discussion below).

A node has satellites if the creating event introduces a function symbol or introduces more than one axiom. For example, the **GROUND-ZERO** node has as its satellites the list of every primitive function symbol, every executable counterpart, and every rule added by **BOOT-STRAP**. **DCL** nodes have a single satellite that is the executable counterpart of the symbol introduced. **ADD-SHELL** nodes have many satellites: the base object, the recognizer, the accessors, the executable counterparts of all the new symbols, and all the rules generated by the shell axioms. No name is a satellite of more than one node.

We say a Lisp symbol is a *citizen* of the data base if it is either the name of a node or is a satellite of a node. The name of each axiom, theorem, function, and

executable counterpart is a citizen. The only other citizens are the names of events such as **TOGGLES**. Henceforth, whenever we refer to a citizen **x** as a node we mean either the node with name **x**, if such a node exists, or the node of which **x** is a satellite. Every citizen has a *status*, **ENABLED** or **DISABLED**, that is used as the status of the associated rule(s).

The *primary* of a citizen, **fn**, is defined as follows. If **fn** is a satellite of **p**, then **p** is the primary of **fn**. Otherwise, **fn** is a node and is its own primary.

The arcs in the data base graph represent “immediate dependency.” The relation is defined below. It is *not* the obvious “logical dependency” relation.

We define *immediate dependency* as follows. Every node in the data base is immediately dependent on **GROUND-ZERO**. An **ADD-SHELL** event is immediately dependent upon the primary of every other shell function involved in its definition, i.e., the older recognizers in the type restrictions and the older base objects in the default values. Every node whose event contains a term (including the terms used in hints) is immediately dependent upon the primary of every function symbol in every such term. Every node whose admissibility requires theorem proving (e.g., **DEFN** and **PROVE-LEMMA** nodes) is immediately dependent upon the primary of every citizen reported used in the proofs. Furthermore, every such node is immediately dependent upon the primary of every prior event that generated a type prescription rule about any function used in the proofs.³⁰ **TOGGLE** events (including **DISABLES** and **ENABLES**) are dependent upon the primary of the event enabled or disabled. **TOGGLE-DEFINED-FUNCTIONS** events are dependent only upon **GROUND-ZERO**.

The above definition establishes the meaning of the “immediate dependents” of every node. Note that the immediate dependents of a node include the reported uses of all of its satellites. We make the convention that the “immediate dependents” of a satellite are just those of its primary. For example, suppose **PUSH** is declared as a constructor with accessors **TOP** and **POP**. Then **PUSH** is the name of a node and **TOP** and **POP** are among its satellites. If **TOP** is used in some subsequent event, the event is recorded among the dependents of **TOP**’s primary, namely **PUSH**. Uses of **POP** and the other functions introduced in the shell declaration are similarly recorded against **PUSH**. We freely mix such dependencies because we do not provide a way to remove one accessor, say, from the data base without removing the entire shell declaration.

³⁰The rewriter does not keep track of the use of type prescription rules because the type set mechanism is so heavily used. This clause in the definition of dependency is an attempt to compensate for this inadequacy by sweeping into the dependency relation some of the events that *might* have been used. Even this does not guarantee to get all of the rules used. See the warning below.

Warning: This notion of “immediate dependency” is flawed. Not every logical dependent is among the “immediate dependents.” The problem is that not all facts *used* by the theorem prover are *reported* used. In particular, type prescription lemmas are not tracked and the executable counterparts for the metafunctions **SUBRP**, **APPLY-SUBRP**, **FORMALS**, **BODY**, **V&C\$**, **V&C-APPLY\$**, **EVAL\$**, **APPLY\$**, and **FOR** do not leave tracks when they operate on quoted constants, even though the values of those metafunctions depend not just on the constants but how certain functions are defined.

For example, suppose **FN** is a user-defined function. Then **(SUBRP 'FN)** reduces to **F**. Furthermore, no record is left that the computation depends on **FN**; indeed, in a way, it doesn't depend on the function **FN**, it depends on the atom **(PACK '(70 78 . 0))**. Thus, by using **UNDO-NAME** to remove an event and its dependents it is possible to get the data base into an inconsistent state: define **FN**, prove **'FN** is not a **SUBRP**, undo **FN**, add a shell with **FN** as the constructor, prove **'FN** is a **SUBRP**. Because we do not track the dependency of **(SUBRP 'FN)** on **FN**, undoing **FN** fails to undo the lemma that **'FN** is a non-**SUBRP**.

Moral: Do not rely upon the validity of any formula proved in a session in which **UNDO-NAME** was used. Think of **UNDO-NAME** as a quick and dirty way to approximate the logical theory obtained by removing the given event. When an interactive session has successfully led to the proof of the main theorem, replay the entire sequence of events from **GROUND-ZERO** or some trusted library without any undoing. See also the discussion of **UNDO-BACK-THROUGH**, a trusted way to remove events from the data base.

We now discuss the **DATA-BASE** procedure.

General Form:

(DATA-BASE query name)

The first argument is a Lisp symbol corresponding to one of the queries listed below. If the second argument, **name**, is not a citizen, the result is **NIL**. Otherwise, the result depends upon the **query** as described below.

query	result
PRIMARY	If name is the name of a node, DATA-BASE returns name . Otherwise, name is the name of a satellite and DATA-BASE returns the name of the node to which name is attached. Since NIL is not an event name, (DATA-BASE 'PRIMARY x) is non- NIL iff x is a citizen. If the result is different than x , x is a satellite of the result.
EVENT-FORM	Returns an event command equivalent to the one the user submitted to create name . name may be a satellite of the

event returned. The command returned will be a list whose first element is the name of an event command, e.g., **PROVE-LEMMA**, **ADD-SHELL**, **TOGGLE**, etc. All list structures used as terms within the command are the result of translating (with **TRANSLATE**) the user type-in. The **EVENT-FORM** returned may differ in other respects from the one typed by the user. For example, the user may type a **DISABLE** command and the **EVENT-FORM** may be an equivalent **TOGGLE**, or the user may have omitted the **hints** argument to a **PROVE-LEMMA** command and the **EVENT-FORM** may supply the **hints NIL**.

- STATUS** Returns the Lisp symbol **DISABLED** if **name** is currently disabled; otherwise returns the symbol **ENABLED**. Recall that if the name of a rule is **DISABLED** the rule is never applied.
- SATELLITES** Returns a list of all the satellites of the primary of **name**.
- FORMULA** Roughly speaking, **DATA-BASE** returns the formula named by **name**. However, this is not a well-defined concept since some names (e.g., the name of a **TOGGLE** event) do not have any associated formula, and other names (e.g., the name of a shell constructor) have many associated formulas. The precise specification is as follows: If **name** is the name of an **ADD-AXIOM**, **CONSTRAIN**, **FUNCTIONALLY-INSTANTIATE** or **PROVE-LEMMA** event (including those generated by **AXIOM** and **LEMMA** commands), the translation of the third argument of that event command is returned (i.e., the axiom added or theorem proved). If **name** is the name of a defined function—even a satellite—the defining equation is returned as a formula of the form **(EQUAL (name x₁ ... x_n) body)**. If **name** is anything else, **NIL** is returned.
- IMMEDIATE-DEPENDENTS** Returns a list containing the names of the immediate dependents of **name**, ordered chronologically.
- ALL-DEPENDENTS** Returns the transitive closure of the immediate dependents relation starting at **name**, ordered chronologically. This is the list of all events that somehow depend upon **name**—in so far as we are able to determine given the incomplete usage reports. It is this list of events, plus **name** itself, that will be removed from the data base if **name** is removed.

IMMEDIATE-SUPPORTERS

Returns the list of all events claiming **name** among their immediate dependents, ordered chronologically.

ALL-SUPPORTERS

Returns the transitive closure of the immediate supporters relation starting at **name**, ordered chronologically. This is the list of all events upon which **name** somehow depends—in so far as we are able to determine given the incomplete usage reports.

ANCESTORS

If **name** is not a function symbol, **NIL** is returned. Otherwise, returns a list of the names of all the **DEFN**, **CONSTRAIN**, **ADD-SHELL** and **DCL** events which introduced the function symbols that are ancestors of **name**. See page 345 for the definition of “ancestor.”³¹

12.16. DCL

General Form:

(DCL fn args)

Example Form:

(DCL LOC (VAR MEM))

DCL is an event command for introducing an undefined function symbol into the logic. It does not evaluate its arguments. **fn** must be a new function symbol and **args** must be a list of **n** distinct variable symbols. **DCL** declares **fn** to be a function symbol of arity **n**. **fn** is henceforth considered “old.” **DCL** adds no axioms to the theory. Unless an error is caused because of improper arguments, **DCL** adds a new event to the data base, prints a time triple, and returns the new event name, **fn**.

If a function symbol has been introduced with **DCL** we say it is an *undefined* function symbol.

The example declaration above declares **LOC** to be a function of arity **2**, provided the name **LOC** has not been previously introduced as an event or function name. After the declaration, it is permitted to use such expressions as **(LOC V M)** and **(LOC (CAAR X) (MEM ST))** as terms.

³¹The Lisp program **ANCESTORS** takes a list of function symbols and returns the union of the ancestral event names of each of them. **(DATA-BASE 'ANCESTORS name)** is just **(ANCESTORS (LIST name))**. When the ancestors of a clique of function symbols is needed, the routine **ANCESTORS** is much more efficient than multiple calls of **DATA-BASE**.

Undefined function symbols can be used in formulas, e.g., axioms, definitions, and theorems. However, they cannot subsequently be defined. Undefined functions are not explicit value preserving: calls of undefined functions on explicit values cannot be reduced to explicit values by the primitive axioms and definitions. Calls of functions defined in terms of undefined functions are usually not explicit value preserving—not if the undefined functions are necessarily involved in the computation. Since **DCL** adds no axioms, the primitive metafunctions **SUBRP**, **APPLY-SUBR**, etc., are undefined on '**fn**' if they were undefined on '**fn**' before the **DCL**. The user may add axioms to characterize **fn**. However, in that case we recommend that you use **CONSTRAIN**, rather than **DCL**, to introduce **fn** and thus avoid the use of **ADD-AXIOM**.

It may be instructive to note that **(DCL fn args)** could be implemented as **(CONSTRAIN fn-INTRO NIL T ((fn (LAMBDA args T))))**, except that the latter creates the event name **fn-INTRO** whereas the former uses **fn** as both the event name and the name of the introduced function.

12.17. DEFN

General Forms:

```
(DEFN fn args body)
and
(DEFN fn args body hints)
```

Example Forms:

```
(DEFN LEN (X)
  (IF (NLISTP X)
    0
    (ADD1 (LEN (CDR X)))))
```

and

```
(DEFN UP (I J)
  (IF (LESSP I J)
    (UP (ADD1 I) J)
    T)
  ((LESSP (DIFFERENCE J I)))))
```

DEFN is an event command for defining a function symbol. It does not evaluate its arguments. **args** must be a list of **n** variable names **v₁**, ..., **v_n**. The command has the same theoretical effect as the axiomatic act:

Definition.

(fn v₁ ... v_n) = body

with the following exception. Admissibility restriction (d) of the principle of definition is that “there is a term **m** ...” such that certain derived formulas are theorems. The *implemented* principle of definition further requires that the theorems be provable by the theorem prover’s simplification process (see page 266)! If admissible, the theory is augmented by the axioms introduced by the above axiomatic act. In addition, the rules corresponding to those axioms are added to the data base. **DEFN** then computes the induction schemes suggested by the various measured subsets (see page 272). It also computes a type prescription rule for the function (see page 282) and preprocesses the body of the function to create a definitional replacement rule (named **fn**) for expanding calls of the function. To preprocess the body, **DEFN** expands the calls of all nonrecursive primitive functions in the body (e.g., **AND**, **FIX**) and then normalizes the resulting **IF**-expression so that no **IF** contains an **IF** in its test. This may exponentially increase the size of the body. Finally, **DEFN** creates a new event named **fn** and links it into the data base. Then **DEFN** prints a narrative describing the justification(s) of the definition and the type prescription computed, a time triple is printed, and **fn** is returned.

If the axiomatic act above is inadmissible (or the simplifier cannot prove the required formulas), an appropriate message is printed, an error is caused, and the data base is left as it was before the **DEFN** command was executed.

We have not yet discussed how the appropriate measure term is selected by **DEFN**. The answer is simple: unless otherwise instructed by the **hints** argument, **DEFN** tries **COUNT** on each argument changed in recursion. The optional **hints** argument permits the user-specification of a measure term to be used.

If non-**NIL**, **hints** must be a list of doublets as described below. Each doublet specifies a well-founded relation and a measure term. For each doublet, **DEFN** generates formulas which, if theorems, are sufficient to establish the theorems described in requirement (d). **DEFN** tries to prove each of the formulas for each doublet. If it is successful for any doublet, the definition is admissible. If it is unsuccessful on each, an error is caused. The **hints** argument allows multiple “justifications” of a definition; for example the first doublet might claim that the **COUNT** of the first argument gets smaller, while the second doublet claims that the second argument gets closer to **100**. Having alternative justifications makes the theorem prover more powerful when it is considering what induction should be used on a given conjecture. See the discussion of the induction process, page 272.

Each doublet must be of the form **(wfn m-term)**, where **wfn** is a “known well-founded relation” and **m-term** is a term all of whose variables are among those in **args**, namely **x₁**, ..., **x_n**. The known well-founded relations depend

upon how **BOOT-STRAP** was called. If booted in **NQTHM** mode, the known well-founded relations are **ORD-LESSP** and **LESSP**.³² In **THM** mode, the known well-founded relations are **LESSP**, **LEX2**, and **LEX3**.

When the system fails to prove the formulas justifying a submitted definition, it prints out an explanatory message that lists the measures tried and the simplified (but unproven) goals generated. If the measure you believe is the correct one was not tried, use the **hints** argument to specify the correct measure. If the measure you believe is the correct one was tried but could not be shown to decrease, prove **REWRITE** rules suitable to establish the printed goals and resubmit the definition.

12.18. DEFTHEORY

General Form:

```
(DEFTHEORY name theory)
```

Example Form:

```
(DEFTHEORY LIST-THEORY
  (APPEND *1*APPEND REVERSE *1*REVERSE
    ASSOC-OF-APPEND APPEND-RIGHT-ID REVERSE-REVERSE))
```

DEFTHEORY is an event command. It does not evaluate its arguments. The first argument, **name**, must be a new name. The second argument, **theory**, must be a nonempty list of citizens. **DEFTHEORY** creates a new event, **name**, that associates **theory** with **name**. Henceforth **name** may be used as a theory name and its “value” is **theory**.

In addition to the theory names defined via **DEFTHEORY** there are two built-in theory names:

- **GROUND-ZERO**: a theory name whose value is the list of citizens introduced by **BOOT-STRAP**, i.e., the satellites of the **GROUND-ZERO** event in the current data base; and
- **T**: a theory name whose value is the list of all citizens in the current data base.

³²Technically, the logic requires use of **ORD-LESSP** to justify a definition or an induction. But it is easy to see, from the definition of **ORD-LESSP**, that if the analogous measure theorems are proved about **LESSP** then the required theorems could be proved for **ORD-LESSP**. When **LESSP** is used as the relation it is unnecessary to prove that the measure term satisfies **ORDINALP**: since **LESSP** coerces non-**NUMBERPs** to 0, the measure term used in the required **ORD-LESSP** justification is just the **FIX** of the measure term used in the **LESSP** justification.

Theory names make it convenient to enable or disable all the names associated with the theory. For example, the event (**DISABLE-THEORY** *name*) will globally disable all of the citizens in the value of the theory named *name*. There is an analogous hint to **PROVE-LEMMA**, also called **DISABLE-THEORY**, that permits the “local disabling” of all the names in a theory during a proof. The analogous enabling command and hint is called **ENABLE-THEORY**. See **DISABLE-THEORY**, **ENABLE-THEORY**, and the hints of **PROVE-LEMMA** for details. It is also possible to map over all the names in the value of a theory and set the enabled/disabled status of each according to the type of event that introduced the name, e.g., one can enable all the definitions and shell axioms and disable all the proved lemmas in a theory. See **SET-STATUS**. Using the built-in theory **T** and **SET-STATUS** it is possible to “erect barricades” in your data base marking the end of one phase of a project and the beginning of another. For example, by disabling all rules except definitions one “wipes the slate clean” and is able to begin developing a new set of rules about selected definitions just as though the definitions had just been introduced—with the added advantage that when old rules are recognized as useful they can be enabled. See **SET-STATUS**.

Theories are very useful in connection with projects involving several layers. Imagine a project in which there are just two layers, which we shall call the “low” level and the “high” level. The low layer is defined in terms of Nqthm primitives and the high level is defined primarily in terms of the low level functions. Let **LOW-DEFNS** be a theory that contains the low level function names and let **HIGH-DEFNS** be the analogous theory for the high level functions. Suppose a large collection of rules has been developed for each layer. Collect the rule names into two theories, **LOW-RULES** and **HIGH-RULES**. The intention is that the rules for a given layer be sufficient for reasoning about the functions at that layer. Thus, to prove a high level theorem, one should have **HIGH-DEFNS** disabled and **HIGH-RULES** enabled, so the high level functions are not expanded into their low level equivalents and the high level rules are employed to manipulate them. Now suppose a useful new high level rule is discovered and that it is not derivable from the current high level rules. To prove such a rule it is necessary to “drop down a level” and expose the definitions of the high level concepts. Thus, **HIGH-DEFNS** should be enabled, **HIGH-RULES** should be disabled and **LOW-RULES** should be enabled. The proof of the new high level rule is constructed from the rules of the lower level. Theories make it convenient to move between such layers or to ensure that the theorem prover stays at a given layer. This is particularly important when, as always seems to happen, the various layers share certain function symbols and contain different sets of rules for them according to the paradigms of use in a given layer. In these cases it is often the intended use rather than the statement of a theorem that indicates what layer is involved.

12.19. DISABLE

General Form:

(DISABLE name)

Example Forms:

(DISABLE TIMES-2-REWRITE)

and

(DISABLE REVERSE)

and

(DISABLE *1*REMAINDER)

DISABLE is an event command that sets the status of a citizen to **DISABLED**. The command does not evaluate its argument. The argument, **name**, must be a citizen. The command creates a new event name, **new-name**, adds a new event to the data base with **new-name** as its name, and sets the status of **name** in the data base to **DISABLED**. Then **DISABLE** prints a time triple and returns **new-name**. **(DISABLE name)** is just an abbreviation for **(TOGGLE new-name name T)**. Indeed, even though we speak of “**DISABLE** events” the data base actually records all **DISABLE**, **ENABLE**, and **TOGGLE** commands as **TOGGLE** events.

Roughly speaking, the effect of **(DISABLE name)** is to prohibit the use of **name** in all subsequent proof attempts until the **DISABLE** is overridden with an **ENABLE** command or temporarily overridden with an **ENABLE** hint in **PROVE-LEMMA** or **new-name** is undone.

More precisely, suppose the status of **name** is **DISABLED** in the data base. Then if **name** is the name of a rule, the rule is never applied. If **name** is the name of a defined function, calls of the function are not expanded.³³ If **name** is the name of the executable counterpart of a function, applications of the function to explicit values in formulas being proved are not reduced to explicit values.³⁴ Note that, while legal, there is no sense in disabling any name other than the name of a rule, a function, or an executable counterpart. For example, disabling the name of a **DISABLE** event has no effect.

³³However, if all the arguments of a function call are explicit values, the executable counterpart of the function may be executed. Thus, **(REVERSE '(1 2 3))** will be replaced by **'(3 2 1)** even if **REVERSE** is disabled. To prevent this, disable the executable counterpart, ***1*REVERSE**, of the function as well.

³⁴Applications of a **DISABLED** executable counterpart are reduced when they arise in the execution of some other executable counterpart. For example, suppose **REVERSE** is defined in terms of **APPEND** and the executable counterpart of **APPEND** is **DISABLED** as by **(DISABLE *1*APPEND)**. Then **(APPEND '(3 2) '(1))** will not be reduced to an explicit value when it appears in formulas being proved. But **(REVERSE '(1 2 3))** does reduce to **'(3 2 1)**, even though it requires the reduction of **APPEND** calls.

The status of a name in the data base is either **ENABLED** or **DISABLED** according to the most recently executed **ENABLE** or **DISABLE** (or equivalent **TOGGLE**) event for that name still in the data base. If no such event is in the data base, the status is **ENABLED**.

A citizen can also be disabled “locally” (without affecting its status in the data base) via the **DISABLE** and **DISABLE-THEORY** hints to **PROVE-LEMMA**, **CONSTRAIN**, and **FUNCTIONALLY-INSTANTIATE**.

Disabling is often important in trying to construct large proofs. Some users prefer to operate in the mode in which all names are disabled globally and a name must be explicitly enabled locally to be used in a proof. In this mode the theorem prover usually responds quickly because its options are so limited. The command **LEMMA** is a variant of **PROVE-LEMMA** that makes it easy to operate in this mode when desired.

Three situations requiring disabling arise so often we will discuss them explicitly here.

12.19.1. A Bolt from the Blue

Scenario: A proof that was supposed to work failed. Upon analysis of the output you see that a key **REWRITE** rule was not applied. Upon further analysis you see that it is because the target term to which it was supposed to apply was deformed by the prior application of another rule.

The most common solution to this is to disable the unwanted rule, either “globally” with the **DISABLE** command or “locally” with the **DISABLE** hint to **PROVE-LEMMA**. Depending on the nature of the unwanted rule it is sometimes more reasonable to re-express the desired rule so that its left-hand side anticipates the action of the other rule. This is particularly true if the “unwanted” rule is one of the fundamental rules of your evolving theory and having it disabled is disastrous. It should be noted that the “unwanted” rule might be a definition which is unexpectedly expanding.

12.19.2. Nonrecursive Definitions

Scenario: A complicated nonrecursive function has been defined, and you have just finished proving a set of beautiful **REWRITE** rules that hide its complexity and let you manipulate calls of the function smoothly. However, none of your rules are ever applied!

The problem here is that nonrecursive functions are (almost) always expanded immediately. Thus, no rule about such a function will ever find a match in a

simplified conjecture. This problem is discussed further on page 398. The solution, generally, is to disable the nonrecursive function as soon as you have characterized it with **REWRITE** rules. If you use nonrecursive functions extensively and do not want all proofs about them to proceed by simply expanding them into their cases, we advise the following approach: define the function, prove the fundamental properties as rewrite rules, disable the function, and base all subsequent proofs on the fundamental properties. You will frequently discover that you omitted a “fundamental property.” **ENABLE** will be useful then.

12.19.3. Hierarchical Rule Development

Scenario: You have just finished proving a fundamental result in your evolving theory. You expect the result to be used widely in subsequent proofs. However, you find that it is not.

The problem here, often, is that the lemmas proved in order to derive the fundamental result are “getting in the way.” Frequently, the proof of a major theorem requires the proofs of many minor ones that handle special cases. These “lemmas” are often formulated haphazardly simply to permit the derivation of the proof of the main theorem. However, because the lemmas necessarily deal with the same clique of function names as the main theorem, they will often find unanticipated applications outside the proof of the main theorem. In particular, they may well deform the very targets for which the main theorem was intended. The standard solution is to disable these intermediate or inconsequential lemmas once the main theorem has been proved.

12.20. DISABLE-THEORY

General Form:

(DISABLE-THEORY theory-name)

Example Forms:

(DISABLE-THEORY LIST-THEORY)

DISABLE-THEORY is an event command. It does not evaluate its argument. The argument, **theory-name**, should be the name of a theory. See **DEFTHEORY**. **DISABLE-THEORY** creates a new event name, **name**, adds a new event to the data base with **name** as its name, and sets the status of every name in the list of citizens denoted by **theory-name** to **DISABLED** unless the

name's status already is **DISABLED**. Then **DISABLE-THEORY** prints a time triple and returns **name**. The **DISABLE-THEORY** hint to **PROVE-LEMMA** permits one to disable a theory “locally.”

(DISABLE-THEORY theory-name) is just an abbreviation for **(SET-STATUS name theory-name ((OTHERWISE DISABLE)))**, which is defined on page 381. The careful reader will note that **theory-name** is therefore also permitted to be an explicit list of citizen names or a dotted-pair naming the endpoints of an interval of citizen names. However, to highlight the symmetry between the global disabling effect of this event and the local disabling effect of the **DISABLE-THEORY** hint to **PROVE-LEMMA**, where the “arguments” must be theory names, we prefer to encourage the view that **DISABLE-THEORY** operates on a named theory. The experienced user may wish to use the more general forms of **DISABLE-THEORY**.

See page 328 for a discussion of the effect of enabling or disabling a name and why it is done. See **DEFTHEORY** for a general discussion of why theories are useful. See the documentation of **SET-STATUS** for a warning about the peculiar interaction between **SET-STATUS** and undoing.

12.21. DO-EVENTS

General Form:

(DO-EVENTS events)

Example Forms:

```
(DO-EVENTS '( (DEFN LEN (X)
               (IF (NLISTP X)
                   0
                   (ADD1 (LEN (CDR X)))))
              (PROVE-LEMMA LEN-APPEND (REWRITE)
               (EQUAL (LEN (APPEND A B))
                       (PLUS (LEN A) (LEN B))))))

(DO-EVENTS XXX)
```

DO-EVENTS executes each event form in a list of such forms and is the most common way to replay an edited sequence of undone events or process an unproblematic sequence of newly created commands. The argument, **events**, is evaluated and should be a list of event forms.

DO-EVENTS iterates down **events**, considering each event form in turn. For each form it prettyprints the form, then evaluates the form (which causes

additional printing, e.g., of proofs and time triples), prints the value of the form, and then prints a form feed.³⁵ **DO-EVENTS** terminates either when all events have been processed or some event either causes an **ERROR** or **FATAL ERROR** or some **PROVE-LEMMA** fails. **DO-EVENTS** returns **T** if all events were successfully processed and **NIL** otherwise.

Normally, the output of **DO-EVENTS**, and of the event commands it executes, is printed to the terminal. However, when used by **PROVEALL**, all output, including error messages, goes to the **proofs** and **err** extensions as noted on page 354. In this case, **DO-EVENTS** indicates its progress through **events** by printing out the name of each event form just before the form is executed. It prints out a comma upon completion of the execution of each form.

12.22. DO-FILE

General Form:

```
(DO-FILE file)
```

Example Form:

```
(DO-FILE "region.events")
```

DO-FILE executes the forms in a file, treating them as though they were Nqthm event forms. The argument is evaluated and should be the name of a file containing Nqthm events. The forms in the file are iteratively read, printed to the terminal, and evaluated. If any form causes an error or returns **NIL** then it is considered to have failed and **DO-FILE** terminates and returns **NIL**. **DO-FILE** terminates and returns **T** when it has processed all of forms in the file.

DO-FILE is similar in spirit to **PROVE-FILE** in that it processes a file presumed to contain events. Unlike **PROVE-FILE** it does not check that the file actually contains events and will execute any Lisp form. In that sense, **DO-FILE** is similar to the Common Lisp function **LOAD**, except that **DO-FILE** prints the forms before execution and aborts if any form fails. Because of its cavalier attitude toward the forms in the file, **DO-FILE** should not be used for the final “endorsing” replay of a script file. But it is very handy in daily work where, for example, it is not uncommon to replay a segment of a larger event file by writing it to a temporary file and calling **DO-FILE**.

³⁵Actually, **DO-EVENTS** prints the value of the global Lisp variable **EVENT-SEPARATOR-STRING** to separate one event and its output from the next. The initial value of that variable is **#\Page**.

12.23. ELIM

ELIM is one of the rule class tokens that may be given to **ADD-AXIOM**, **CONSTRAIN**, **FUNCTIONALLY-INSTANTIATE** or **PROVE-LEMMA** to indicate what kinds of rules should be generated from a formula. **ELIM** rules are used in the destructor elimination process, which is the second process tried (simplification being the first). We describe the destructor elimination process on page 269. On page 293 we describe how **ELIM** rules are generated and how they are used. On page 412 we give some hints regarding the use of **ELIM** rules.

12.24. ENABLE

General Form:

(ENABLE name)

Example Forms:

(ENABLE TIMES-2-REWRITE)

and

(ENABLE REVERSE)

and

(ENABLE *1*REMAINDER)

ENABLE is an event command that sets the status of a citizen to **ENABLED**. The command does not evaluate its argument. The argument, **name**, must be a citizen. The command creates a new event name, **new-name**, adds a new event to the data base with **new-name** as its name, and sets the status of **name** in the data base to **ENABLED**. Then **ENABLE** prints a time triple and returns **new-name**. **(ENABLE name)** is just an abbreviation for **(TOGGLE new-name name NIL)**.

See the discussion of **DISABLE** for details.

A citizen can also be enabled “locally” (without affecting its status in the data base) via the **ENABLE** and **ENABLE-THEORY** hints in **PROVE-LEMMA**, **CONSTRAIN**, and **FUNCTIONALLY-INSTANTIATE**.

12.25. ENABLE-THEORY

General Form:

(ENABLE-THEORY theory-name)

Example Forms:

(ENABLE-THEORY LIST-THEORY)

ENABLE-THEORY is an event command. It does not evaluate its argument. The argument, **theory-name**, should be the name of a theory. See **DEFTHEORY**. **ENABLE-THEORY** creates a new event name, **name**, adds a new event to the data base with **name** as its name, and sets the status of every name in the list of citizens denoted by **theory-name** to **ENABLED** unless the name's status already is **ENABLED**. Then **ENABLE-THEORY** prints a time triple and returns **name**. The **ENABLE-THEORY** hint to **PROVE-LEMMA** permits one to enable a theory "locally."

(ENABLE-THEORY theory-name) is just an abbreviation for **(SET-STATUS name theory-name ((OTHERWISE ENABLE)))**, which is defined on page 381. The careful reader will note that **theory-name** is therefore also permitted to be an explicit list of citizen names or a dotted-pair naming the endpoints of an interval of citizen names. However, to highlight the symmetry between the global enabling effect of this event and the local enabling effect of the **ENABLE-THEORY** hint to **PROVE-LEMMA**, where the "arguments" must be theory names, we prefer to encourage the view that **ENABLE-THEORY** operates on a named theory. The experienced user may wish to use the more general forms of **ENABLE-THEORY**.

See page 328 for a discussion of the effect of enabling or disabling a name and why it is done. See **DEFTHEORY** for a general discussion of why theories are useful. See the documentation of **SET-STATUS** for a warning about the peculiar interaction between **ENABLE-STATUS** and undoing.

12.26. Errors

Each error condition checked and reported by the system is classified into one of three classes:

- **WARNING:** a cautionary message concerning a legal but perhaps unintended aspect of a command, e.g., that the body of a function definition makes no reference to one of the formal parameters.
- **ERROR:** a violation of the preconditions on a command, such as the submission of an ill-formed term, the submission of an inadmissible function definition, the attempted introduction of an event name already in the data base, etc. Generally **ERRORS** can be fixed by editing your type-in, though they sometimes indicate a deeper problem than mere syntax.

- **FATAL ERROR**: a violation of the internal invariants assumed by our code or the exhaustion of some resource (e.g., the variable symbols that may be introduced by destructor elimination). It is generally not possible to fix such an error. Except for resource errors or situations in which the user has entered our code via some route other than the documented commands, **FATAL ERRORs** should not arise.

The system may cause Lisp resource errors, such as stack overflow due to excessively deep function calling (usually an indication of circularities in the rewrite rules you have added) or the exhaustion of dynamic storage space (usually caused by circular rewriting that enlarges the term, excessive case splitting, or combinatoric explosion in the normalization of propositional expressions). The system should not cause any other kind of error, e.g., “Trap: Argument given to CAR was not a list, locative, or NIL,” while it is being used entirely in accordance with this handbook. If such errors occur, please notify us.

All three types of errors print supposedly self-explanatory messages on the terminal. After **WARNINGS** the computation proceeds as though nothing had happened. The other two kinds of errors cause interactive Lisp breaks.

It is not possible to fix an error from within the Lisp break and proceed. Proceeding from the break automatically aborts the computation that led to the error. The only reason we enter a break at all is to give you the opportunity to inspect the state of the computation before it is lost by the abort.

Getting out of the break is a system-dependent operation.

- In Lucid on a Sun, type **:A** to the break prompt.
- In GCL or CMU Common Lisp on a Sun, type **:Q** to the break prompt.
- In Allegro on a Sun, type **:RES** to the break prompt.

These actions should return you to the top level of the Lisp system.

The data base is not disturbed by any command that halts with an **ERROR**. The same cannot be said if a **FATAL ERROR** occurs. See however the discussion of the data base integrity on page 296.

To fix an error you must edit (or retype) the command that caused it. Most Lisp systems support this, and since our code runs in many different implementations of Common Lisp and under many different kinds of host operating systems, we have not tried to implement a uniform error recovery mechanism.

12.27. Event Commands

The *event commands* are those that create nodes in the data base. The event commands are

- **ADD-AXIOM** (with its variant **AXIOM**)
- **ADD-SHELL**
- **BOOT-STRAP**
- **CONSTRAIN**
- **DCL**
- **DEFN**
- **DEFTHEORY**
- **FUNCTIONALLY-INSTANTIATE**
- **PROVE-LEMMA** (with its variant **LEMMA**)
- **SET-STATUS** (with its variants **DISABLE-THEORY** and **ENABLE-THEORY**)
- **TOGGLE** (with its variants **DISABLE** and **ENABLE**)
- **TOGGLE-DEFINED-FUNCTIONS**

The “variants” of the event commands are defined in terms of the basic commands but provide convenient commonly used forms. Thus, for example, a **DISABLE** command actually executes a certain call of **TOGGLE**. It is the **TOGGLE** command that is stored in the data base.

12.28. EVENTS-SINCE

General Form:

(EVENTS-SINCE name)

Example Form:

(EVENTS-SINCE 'REVERSE)

The argument, **name**, is evaluated and must be the name of an event. **EVENTS-SINCE** returns the list of event forms for all events created since **name**. The list includes the event form for **name** itself and is ordered chronologically, starting with the event form for **name**.

12.29. Executable Counterparts

Corresponding to every function symbol in the logic is a Lisp procedure called the *executable counterpart* of the function. The name of the executable counterpart is obtained by concatenating the string “*1*” onto the front of the function symbol. Thus, ***1*APPEND** is the executable counterpart of **APPEND**.

Executable counterparts are used to compute the value of functions on constants. Such computations are the essence of the execution facility provided by **R-LOOP** but also arise in the theorem prover proper. Whenever a term of the form $(\mathbf{fn} \ 't_1 \ \dots \ 't_n)$, e.g., the application of a function to explicit values, is produced, we use the executable counterpart of **fn** to compute the equivalent explicit value, if any.³⁶ To explain further we must first discuss the notion of explicit value.

Recall that an *explicit value* is a variable-free term involving only shell constructors and base objects with the property that each argument of each constructor satisfies the type restriction of the corresponding argument position. For example, $(\mathbf{CONS} \ (\mathbf{ADD1} \ (\mathbf{ZERO})) \ (\mathbf{ZERO}))$ is an explicit value term.

Internally, each explicit value term, **t**, is represented by a Lisp data structure of the form $(\mathbf{QUOTE} \ \mathbf{evg})$, where **evg** is called the *explicit value guts* of **t**. The explicit value guts is just the Lisp object corresponding to what we called the “explicit value descriptor” in section 4.7.5, page 158.³⁷

The explicit value above is represented by $(\mathbf{QUOTE} \ (1 \ . \ 0))$. The Lisp object $(1 \ . \ 0)$ is the explicit value guts of $(\mathbf{CONS} \ (\mathbf{ADD1} \ (\mathbf{ZERO})) \ (\mathbf{ZERO}))$.

Suppose that $'e_1, \dots, 'e_n$ are **n** explicit value terms and that their respective explicit value guts are e_1, \dots, e_n . Suppose **fn** is an **n**-ary function. Then $(\mathbf{fn} \ 'e_1 \ \dots \ 'e_n)$ is a term which is likely to be equal to some explicit value. The executable counterpart of **fn**, ***1*fn**, can sometimes be used to compute the explicit value equivalent to $(\mathbf{fn} \ 'e_1 \ \dots \ 'e_n)$ as follows: Apply the Lisp procedure ***1*fn** to the **n** Lisp objects e_1, \dots, e_n . If a result, **e**, is returned

³⁶More precisely, the executable counterpart of **fn** is so used, provided either **fn** is a shell constructor or base function symbol or it is enabled. Executable counterparts can be disabled individually with **DISABLE** or **TOGGLE**, as in $(\mathbf{DISABLE} \ *1*\mathbf{APPEND})$, in theories with **DISABLE-THEORY**, or in regions of the **CHRONOLOGY** with **SET-STATUS**. All executable counterparts can be disabled with **TOGGLE-DEFINED-FUNCTIONS**. Note that shell constructors and base function cannot be effectively disabled; they are used whether disabled or not. The theorem prover’s internal form requires that explicit values be kept in a certain normal form.

³⁷This is not quite true. A “dotted s-expression” in our formal presentation is a sequence whose next-to-last element is the dot character. In Lisp, the explicit value guts corresponding to a “dotted s-expression” is in fact a binary tree of Lisp list cells whose right-most cell contains an atom other than **NIL**. Such trees are printed by Lisp with a dot. Our dotted s-expressions are read by Lisp and converted into such trees.

then $'e$ is the internal representation of an explicit value term equivalent to $(fn\ 'e_1\ \dots\ 'e_n)$. Otherwise, the execution of $*1*fn$ will signal failure (by executing a Lisp **THROW**). Failure is signaled either because an undefined function is encountered on the execution path or, for functions involving **V&C\$**, certain resource limits are exhausted. See **REDUCE-TERM-CLOCK**.

Note that we have not said that $*1*fn$ computes an equivalent explicit *if there is one*, merely that *if $*1*fn$ computes a value* it represents an equivalent explicit value. However, for a very large class of functions the equivalent explicit value always exists and the executable counterparts compute them.

We say a function **fn** is *explicit value preserving* if either (a) it is one of the primitive functions other than the metafunctions **SUBRP**, **APPLY-SUBR**, **FORMALS**, **BODY**, **V&C\$**, **V&C-APPLY\$**, **EVAL\$**, **APPLY\$**, and **FOR** or (b) it is a defined function with the property that every function symbol used in the body, other than **fn** itself, is explicit value preserving. Explicit value preserving functions have the property that for every application to explicit values there exists an equivalent explicit value. Except for those functions which use the metafunctions excluded above and those which use functions introduced with **DCL** or **CONSTRAIN**, every function in our system is explicit value preserving. It is in this sense we used the word “likely” above. For all explicit value preserving functions, the executable counterparts always compute the corresponding explicit values.

However, in addition to the syntactically defined class of explicit value preserving functions, there are defined functions which in fact reduce, or often reduce, to explicit values on explicit values even though metafunctions or undefined functions are used on some branches through their definitions. **V&C\$** itself is a good example: on many applications to explicit values, e.g., on applications to the quotations of terms composed of explicit value preserving functions, **V&C\$** can be reduced computationally to an explicit value.

Every function symbol in the logic has an executable counterpart whether the function is explicit value preserving or not. The **ADD-SHELL** command and the **DEFN** command create the executable counterparts for the symbols they introduce. The **DCL** and **CONSTRAIN** commands create the trivial executable counterpart (“signal failure”) for their function symbols. The command **TOGGLE-DEFINED-FUNCTIONS** can be used to disable all executable counterparts other than shell constructors and base objects. If $*1*fn$ is disabled then $(fn\ 'e_1\ \dots\ 'e_n)$ is not reduced to an explicit value by execution of $*1*fn$.

It is possible to compile executable counterparts. This generally makes **R-LOOP** evaluate your input much faster, and it will speed up your proofs if they involve explicit values. See **COMPILE-UNCOMPILED-DEFNS**.

12.30. Explicit Values

Explicit values are the canonical constants in the logic: terms composed entirely of shell constructors and bottom objects with the additional property that every subterm satisfies the type restriction for the constructor argument position occupied. See page 137, in Chapter 4 for a formal definition of explicit values. See page 158 for a formal description of the notation in which they are written. See the discussion of “executable counterparts,” page 337, for a discussion of how we compute on explicit values.

12.31. Extensions

See the discussion of File Names on page 339.

12.32. FAILED-EVENTS

General Form:

FAILED-EVENTS

FAILED-EVENTS is a Lisp variable that contains all of the event commands submitted in the current session that did not terminate successfully.

12.33. File Names

Generally speaking it is not necessary to know anything about file names the first few times you use the theorem prover: commands are typed to Lisp at the terminal and their output is printed at the terminal. The first use of file names usually occurs when you want to save the current data base to a file, with **MAKE-LIB**, and subsequently restore the data base, with **NOTE-LIB**. The other common use of file names is at the successful conclusion of a major phase of a project when you will probably want to create an “endorsed data base” (see page 361) and a coherent record of the event commands that construct it and the proofs concerned. This is done by calling the function **PROVE-FILE** or the function **PROVEALL**.

To use these more sophisticated commands it is necessary to understand file name formats and our file name conventions. The file name format used by the theorem prover is that defined by the host operating system. *We assume you are familiar with the file name formats on your machine.* We make two assumptions

about file name formats and then proceed to define our file name conventions. Our assumptions are

- File names are written as Lisp strings, i.e., ASCII character sequences enclosed in double quotation marks, as in `"doe:>nqthm>demo.lib"`.
- A "full file name" is composed of two parts, the "root name" and the "extension." The root name is separated from the extension by the ASCII dot character, ".". The format of both the root name and the extension are dictated entirely by the host file system.

In the development of a given subject, e.g., the proof of correctness of the program **FSTRPOS**, the command file, the library files, the proof file, and the error message file all generally have the same root name, which on a Lisp machine might be `"doe:>smith>fstrpos"` and on a Unix system `"/usr/smith/fstrpos"`.

We use the extension of each file name to indicate the particular kind of data contained in the file. Commands that create or read files, such as **NOTE-LIB** and **PROVE-FILE**, take root names as their arguments and extend them by tacking on one of several fixed extension names according to the type of file to be read or written.

For example, proofs generated by **PROVEALL** are written into a file with the **proofs** extension while error messages are written to the **err** extension. There are, in all, nine different extensions used by the system. The names used above, **proofs** and **err**, are just "generic" names used in this handbook. The actual strings used to form extensions may be specified by the user, by setting each of six global Lisp variables. Thus, the **proofs** extension of `"/usr/smith/fstrpos"` might be `"/usr/smith/fstrpos.proofs"` on one system and `"doe:>smith>fstrpos.prf"` on another. We make this precise in the definition below.

Given a root name, **file**, we speak of nine different *extensions* of **file**, each known by a generic name. Each extension is formed by concatenating **file**, a dot, and then a character string (or Lisp symbol) associated with the generic name. To each generic name there corresponds a Lisp variable whose value is the associated character string. These variables may be set by the user. The generic names, the kind of data contained in such extensions, the Lisp variable concerned, and its initial setting are described below.

events The **PROVE-FILE** program reads and executes the forms in the **events** extension of its **file** argument. The name used for the **events** extension is the value of the Lisp variable **FILE-EXTENSION-EVENTS**, which is initially `"events"`.

- proofs** The **PROVEALL** program opens a file with the **proofs** extension to contain a transcript of the event sequence being executed. In particular, each event form is printed into the **proofs** extension, and then the form is executed with all of its output written to the **proofs** extension—error messages, comments on definitions, proofs, time triples, etc.—and then the value of the completed form is printed into the **proofs** extension. See **PROVEALL** for the details. The name used for the **proofs** extension is the value of the Lisp variable **FILE-EXTENSION-PROOFS**, which is initially **"proofs"**. The variant of **PROVE-FILE** that redirects terminal output, **PROVE-FILE-OUT**, also opens a **proofs** extension of its **file** argument and writes the transcript to it.
- err** The **PROVEALL** program opens a file with the **err** extension to contain all **WARNING**, **ERROR** and **FATAL ERROR** messages generated during the **PROVEALL**. (These messages are also printed into the **proofs** extension as part of the transcript.) The **err** extension contains nothing but such messages and is a convenient summary of the anomalies in the event sequence. The name used for the **err** extension is the value of the Lisp variable **FILE-EXTENSION-ERR**, which is initially **"err"**.
- lib** The **lib** extension is one of two extensions in which the data base is saved by **MAKE-LIB** and restored by **NOTE-LIB**. The other is the **lisp** extension below. The data in the **lib** extension is written from and read into Lisp variables and property lists. The name used for the **lib** extension is the value of the variable **FILE-EXTENSION-LIB** and is initially **"lib"**.
- lisp** The **lisp** extension is used in connection with files that contain Common Lisp code. **MAKE-LIB** and **NOTE-LIB** use the the **lisp** extension for a file containing the Common Lisp definitions of the executable counterparts of the functions in the saved data base. The **lisp** extension is used by **COMPILE-UNCOMPILED-DEFNS** when creating a file to be compiled. These uses of the **lisp** extension concern files created by the system in connection with executable counterparts. The source files of the theorem prover itself are also Common Lisp files and, as noted in Chapter 14, the **lisp** extension is also used by the programs **COMPILE-NQTHM** and **LOAD-NQTHM**. The name used for the **lisp** extension is the value of the Lisp variable **FILE-EXTENSION-LISP**, initially **"lisp"**.

bin

A file with the **bin** extension should be the result of compiling the corresponding **lisp** extension. **COMPILE-UNCOMPILE-DEFNS** uses the **bin** extension to load the compiled file it creates. During installation of our system, **COMPILE-NQTHM** and **LOAD-NQTHM** (see Chapter 14) use the **bin** extension to load compiled files. The name used for the **bin** extension is the value of the Lisp variable **FILE-EXTENSION-BIN**. In general, **FILE-EXTENSION-BIN** should be set to the same extension used by the local compiler. That is, if your compiler puts the object code for file **foo** into **foo.o** then **FILE-EXTENSION-BIN** should be set to **"o"**; if your compiler puts the object code for file **foo** into **foo.bin** then **FILE-EXTENSION-BIN** should be set to **"bin"**. However, the only use we make of **FILE-EXTENSION-BIN** is when one of the programs mentioned above **LOADs** the compiled code for a given root name. As far as we are aware, all implementations of the Common Lisp **LOAD** program now support the convention that, when no extension is used and a compiled file for the given root name exists, then the compiled file is **LOADed**. Thus, it is not necessary to know the extension used by the compiler—which is to say, one need not extend the root name to refer explicitly to the compiled file to be loaded. We therefore adopt the convention that when **FILE-EXTENSION-BIN** is **NIL** no extension is tacked on. This is the initial setting for **FILE-EXTENSION-BIN**. If your implementation of Common Lisp does not support the convention, you should set **FILE-EXTENSION-BIN** to the same string used by the local compiler.

started

The **PROVE-FILE** program, when observing its flag file protocol, signals the commencement of processing by creating the **started** extension of its **file** argument. The name used for the **started** extension is the value of the variable **FILE-EXTENSION-STARTED** and is initially **"STARTED"**.

proved

The **PROVE-FILE** program, when observing its flag file protocol, signals the successful termination of processing by creating the **proved** extension of its **file** argument. The name used for the **proved** extension is the value of the variable **FILE-EXTENSION-PROVED** and is initially **"proved"**.

fail

The **fail** extension is a file generated by **PROVEALL** and **PROVE-FILE** (when it is observing its flag file protocol)

when the event sequence causes an error or a proof fails. The name used for the **fail** extension is the value of the Lisp variable **FILE-EXTENSION-FAIL** and is initially **"fail"**.

We now raise a further complication in the use of file names: most Lisps—or the underlying operating systems—provide default mechanisms by which our “full file names” are further elaborated before a unique file identifier is generated. For example, on Unix the user who is “connected” to the directory **/usr/smith** may find that the **lisp** extension of the root name **"fstrpos"**, namely **"fstrpos.lisp"**, actually identifies the file **"/usr/smith/fstrpos.lisp"**. Similarly, by setting the appropriate “default path names,” the user on the Lisp Machine can arrange for a simple root name such as **"fstrpos"** to identify files with much more elaborate unique names.

The Lisp variable ***DEFAULT-NQTHM-PATH*** may be used to determine the directory from which files are read and written while compiling, loading, or using the theorem prover. The initial value of ***DEFAULT-NQTHM-PATH*** is **NIL**, which means “do nothing.” However, if the value is non-**NIL**, then we merge that value, using the Common Lisp function **MERGE-PATHNAMES**, whenever we **OPEN** a file for reading or writing and whenever we **LOAD** a file. If you specify a full pathname when using functions such as **NOTE-LIB**, **MAKE-LIB**, and **PROVEALL**, then the value of ***DEFAULT-NQTHM-PATH*** is irrelevant. However, if you specify a non-**NIL** value for ***DEFAULT-NQTHM-PATH*** and you specify no directory in a file name passed to **NOTE-LIB**, **MAKE-LIB**, etc., then the file used will be on the directory specified by ***DEFAULT-NQTHM-PATH***. When non-**NIL**, ***DEFAULT-NQTHM-PATH*** should be set to a string (not a pathname) that gives the full name of the relevant directory. It is important that the last character of the string be the character that separates components of a pathname on your system, e.g., slash on a Unix system or colon on a Macintosh. Example values of ***DEFAULT-NQTHM-PATH*** are **"/usr/smith/"** and **"doe:>nqthm>"**.

Finally, when Nqthm opens a file for output it explicitly deletes any existing file by that name.

12.34. FUNCTIONALLY-INSTANTIATE

General Forms:

```
(FUNCTIONALLY-INSTANTIATE name rule-classes term
                           old-name fs)
```

and

```
(FUNCTIONALLY-INSTANTIATE name rule-classes term
                           old-name fs hints)
```

Example Form:

```
(FUNCTIONALLY-INSTANTIATE TIMES-PR-IS-TIMES-AC
  (REWRITE)
  (EQUAL (TIMES-AC L Z) (TIMES-PR L Z))
  H-PR-IS-H-AC
  ((H TIMES)
   (H-PR TIMES-PR)
   (H-AC (LAMBDA (X Y) (TIMES-AC X Y)))))
```

FUNCTIONALLY-INSTANTIATE is like **PROVE-LEMMA** in that it proves a theorem, **term**, and adds it to the data base as a theorem under the name **name** and as rules of the classes specified by **rule-classes**; however, **term** must be the functional instantiation under **fs** of some already proved theorem, named **old-name**. Rather than proving **term** directly, **FUNCTIONALLY-INSTANTIATE** attempts to verify that the axioms used in the proof of **old-name** are theorems under **fs**.

FUNCTIONALLY-INSTANTIATE is an event command. It does not evaluate its arguments. **name** must be a new name, **rule-classes** must be a (possibly empty) list of rule classes, and **term** must be a term or the word ***AUTO***. If **old-name** is a symbol, then the **FORMULA DATA-BASE** query for **old-name** must be some non-NIL formula **old-term**.³⁸ **fs** must be a functional substitution; the domain of **fs** must consist of function symbols already introduced into the theory by the user via **CONSTRAIN**, **DCL**, or **DEFN** (i.e., the domain may not include symbols in the **GROUND-ZERO** theory nor functions introduced with **ADD-SHELL**) and the arity of each function in the domain of **fs** must be the arity of the corresponding function in the range. Unless **term** is ***AUTO***, **term** must be the functional instantiation of **old-term** under **fs**, i.e., **old-term****fs**. (If **term** is ***AUTO***, we treat it as though that were an abbreviation for **old-term****fs** below.) Finally, **hints**, if supplied, must be as required by the **PROVE-LEMMA** event (see page 364).

³⁸The case in which **old-name** is not a symbol is rarely used and we discuss it later.

If the above syntactic restrictions are satisfied, **FUNCTIONALLY-INSTANTIATE** undertakes to prove **term** via functional instantiation. In particular, the theorem prover is called (with the **hints** provided) on the conjunction of the **fs** functional instances of the formulas of certain **ADD-AXIOM**, **CONSTRAIN**, and **DEFN** events. Such an event is involved iff (a) it uses as a function symbol some symbol in the domain of **fs** and (b) it is either (i) an **ADD-AXIOM** or (ii) a **CONSTRAIN** or **DEFN** that introduces a function symbol “ancestral” in **old-term** or some **ADD-AXIOM**. The user of the system need not understand what we mean by “ancestral” since the system determines the relevant formulas and prints them out before undertaking to prove them. However, we define the notion below. If any of the formulas to be instantiated use as a variable some variable that is free in any **LAMBDA** expression in the range of **fs**, then **FUNCTIONALLY-INSTANTIATE** aborts with an error and does not change the data base. Such aborts can always be avoided by choosing new variable names in **fs**. If the proof attempt is successful, **term** is stored in the data base as a theorem and rules of the classes specified by **rule-classes** are generated and stored as well. The name of each rule is **name**. If the proof attempt fails, **FUNCTIONALLY-INSTANTIATE** behaves as does **PROVE-LEMMA**: the “failure banner” is printed along with a time triple and **NIL** is returned.

A function symbol **fn** is *ancestral* in a term if and only if **fn** is an “ancestor” of some symbol used as a function symbol in the term. We say that **fn** is an *ancestor* of **g** if and only if **g** was introduced by a **DEFN** or **CONSTRAIN** event, **ev**, and either **fn** is one of the function symbols introduced by **ev** (including **g** itself) or **fn** is an ancestor of a function symbol used in the axiom added by **ev**. That this definition of ancestral is sufficient to justify functional instantiation is the subject of the main proof in [19].

We wish to make it convenient to apply the same functional substitution to several different theorems in a sequence of **FUNCTIONALLY-INSTANTIATE** events, without having to prove the same constraints repeatedly. Therefore, **FUNCTIONALLY-INSTANTIATE** does not bother to prove the **fs** instance of a formula if any previous **FUNCTIONALLY-INSTANTIATE** did prove it. This requires searching through a record kept of the proof obligations of all previous **FUNCTIONALLY-INSTANTIATE** events. If there are many such events you may wish to limit the search to some particular set of past events. This can be done by using a non-symbol for **old-name**. If **old-name** is not a symbol then it must be a list of the form (**old-name'** **name**₁ ... **name**_k), where **old-name'** is a name whose **FORMULA DATA-BASE** query produces the non-**NIL** formula **old-term** in the foregoing description, and each **name**_i is the name of a previous **FUNCTIONALLY-INSTANTIATE** event whose proof obligations are to be searched. The search is limited to the **name**₁.

12.35. Functional Substitution

General Form:

$((f_1 \ g_1) \ \dots \ (f_n \ g_n))$

Example Form:

`((H PLUS) (G (LAMBDA (X) (TIMES K X))))`

As defined formally on page 183, a “functional substitution” is a function on a finite set of function symbols such that for each pair, $\langle f, g \rangle$, in the substitution, g is a function symbol or **LAMBDA** expression and the arity of f is the arity of g . In our implementation, such substitutions are represented by lists of doublets as shown in the general form above. Each f_i must be a symbol and each g_i must be a function symbol or **LAMBDA** expression of the form `(LAMBDA ($a_1 \ \dots \ a_n$) body)`, where the a_i are distinct variable symbols and **body** is a term. The set of the f_i is called the domain of the substitution and the set of the g_i is called the range. Functional substitutions have two distinct uses in our implementation and each use puts some additional restrictions on the substitution.

Functional substitutions are used with the **CONSTRAIN** event to supply both arities and witnesses for the functions being introduced. In such use, the f_i above must be new names and the **LAMBDA** expressions occurring among the g_i may not contain free variables; **CONSTRAIN** uses the g_i as witnesses for the f_i while verifying that the axiom to be added is satisfiable and then **CONSTRAIN** introduces the f_i as function symbols with the same arities as the corresponding g_i . The arity of `(LAMBDA ($a_1 \ \dots \ a_n$) body)` is n .

Functional substitutions are also used with the **FUNCTIONALLY-INSTANTIATE** event, where they are used to specify how some theorem is to be functionally instantiated. In that usage, the f_i are required to be function symbols already introduced into the theory by the user via **CONSTRAIN**, **DCL**, or **DEFN** (i.e., the f_i may not be **GROUND-ZERO** functions nor functions introduced with **ADD-SHELL**) and the arity of f_i must be the arity of the corresponding g_i .

12.36. GENERALIZE

GENERALIZE is one of the rule class tokens that may be given to **ADD-AXIOM**, **CONSTRAIN**, **FUNCTIONALLY-INSTANTIATE** or **PROVE-LEMMA** to indicate what kinds of rules should be generated from a formula. **GENERALIZE** rules are used in the generalization process, which is the fourth process tried

(after simplification, destructor elimination, and use of equalities). We describe the generalization process on page 271. On page 294 we describe how **GENERALIZE** rules are generated and how they are used.

12.37. Hints to DEFN

General Form:

```
((relation-name measure-term)
...)
```

Example Form:

```
((LESSP (DIFFERENCE (ADD1 MAX) I)))
```

See page 325 for details on the form taken by the **hints** argument to **DEFN**. The **hints** argument to **PROVE-LEMMA**, **CONSTRAIN** and **FUNCTIONALLY-INSTANTIATE** takes a different form, described on page 364.

12.38. Hints to PROVE-LEMMA, CONSTRAIN and FUNCTIONALLY-INSTANTIATE

General Form:

```
((USE (name (var term) ... (var term))
...
(name (var term) ... (var term)))
(EXPAND term ... term)
(DISABLE name ... name)
(ENABLE name ... name)
(DISABLE-THEORY theory-name ... theory-name)
(ENABLE-THEORY theory-name ... theory-name)
(HANDS-OFF fn ... fn)
(INDUCT term)
(DO-NOT-INDUCT T)
(DO-NOT-GENERALIZE T)
(NO-BUILT-IN-ARITH T))
```

The “general form” shown above is meant merely to be suggestive of the form taken by the **hints** argument to **PROVE-LEMMA**, **CONSTRAIN** and **FUNCTIONALLY-INSTANTIATE**. See page 364 for details. The **hints** argument to **DEFN** has a different form and is described on page 325.

12.39. LEMMA

General Forms:

```
(LEMMA name rule-classes term)
and
(LEMMA name rule-classes term hints)
```

Example Form:

```
(LEMMA LEN-APP (REWRITE)
  (EQUAL (LEN (APP A B)) (PLUS (LEN A) (LEN B))))
((ENABLE LEN APP)))
```

Some users prefer to operate in a mode in which all rules are disabled and a name must be explicitly enabled locally to be used in a proof.³⁹ This mode has the advantage of making the theorem prover respond quickly to most challenges, because there are usually so few options available to it. It also allows one to build up very large sets of rules without so much concern about how they act in concert. The disadvantage is that the user must “sketch” each proof by listing the rules involved. Theories are important in this connection because they allow groups of “harmonious” rules to be conveniently enabled in unison. Users often develop several theories corresponding to different situations that commonly arise in their project and then attack a given problem with the theories felt most appropriate.

The **LEMMA** command makes operation in this mode convenient. The form **(LEMMA name rule-classes term)** is just an abbreviation for

```
(PROVE-LEMMA name rule-classes term
  ((ENABLE-THEORY GROUND-ZERO)
   (DISABLE-THEORY T))).
```

That is, when used without a **hints** argument **LEMMA** is like **PROVE-LEMMA** except that all non-primitive rule names are disabled. Little of interest can be proved by **LEMMA** when no hints are given. When a hint argument is supplied to **LEMMA**, a possibly modified version of it is passed to **PROVE-LEMMA**. See page 364 for a description of the **hints** argument to **PROVE-LEMMA**. Roughly speaking, the modified hint disables everything except the names specifically enabled by the user-supplied hint. Let **hints** be the user-supplied hint. We describe below the modified hint argument supplied to **PROVE-LEMMA**, **hints'**.

³⁹This is sometimes called “Bevier-mode” after Bill Bevier who first used it extensively in [3].

- If **hints** contains no **DISABLE-THEORY** or **ENABLE-THEORY** hints, then **hints'** is obtained from **hints** by adding **(ENABLE-THEORY GROUND-ZERO)** and **(DISABLE-THEORY T)**.
- If **hints** contains any **DISABLE-THEORY** hint or an **(ENABLE-THEORY T)** hint, then **hints'** is **hints**. That is, if the supplied hints disable any theories or enable the universal theory, then **LEMMA** is just **PROVE-LEMMA**.
- Otherwise, **hints** contains an **ENABLE-THEORY** hint (but not **(ENABLE-THEORY T)**) and does not contain a **DISABLE-THEORY** hint. Then **hints'** is obtained from **hints** by modifying the **ENABLE-THEORY** entry to include **GROUND-ZERO** (unless it is already there) and adding the hint **(DISABLE-THEORY T)**.

Note that it is possible to operate in both modes in the same session. Most users try to maintain a coherent and effective global data base of rules, using the global theory commands to keep the “right” set of rules enabled for each phase of the project. But when more control is needed, **LEMMA** can be used.

By studying the modified hints **LEMMA** passes to **PROVE-LEMMA** (e.g., by reading the description above or by using **PPE** to print the **PROVE-LEMMA** event generated by a sample **LEMMA** command), the user can see how to achieve the same effect in the less commonly used commands **CONSTRAIN** and **FUNCTIONALLY-INSTANTIATE**.

12.40. MAINTAIN-REWRITE-PATH

General Form:

(MAINTAIN-REWRITE-PATH flg)

Example Form:

(MAINTAIN-REWRITE-PATH T)

This routine turns on and off the maintenance of the rewrite path (see page 307 for a discussion of rewrite paths and see the entries for **ACCUMULATED-PERSISTENCE**, **BREAK-LEMMA**, and **BREAK-REWRITE** for related features). The argument, **flg**, is evaluated and should be either **T** or **NIL**, determining whether the rewrite path is, or is not, henceforth maintained. Maintenance of the stack involves storing into a data structure certain information about every call of the rewriter and degrades the performance to about 60% of its normal speed in proofs involving heavy use of replacement rules. Nevertheless, we frequently maintain the rewrite path because of the information it provides us.

BREAK-LEMMA calls **MAINTAIN-REWRITE-PATH** the first time a rule is monitored since the path must be maintained in case an interactive break occurs. **UNBREAK-LEMMA** does *not* disable path maintenance when the last rule is removed from the list of monitored lemmas. It merely prints a message that path maintenance is still enabled. The reason is that you may still be planning to use **BREAK-REWRITE** or **ACCUMULATED-PERSISTENCE**. If not, you can regain the normal efficiency by turning off the maintenance of the rewrite path. If path maintenance is disabled by the user while monitors are still installed on some lemma names, then a break will occur when those lemmas are activated, but path information will not be available.

The rewrite path is not the Lisp control stack but a separate data structure. It persists even after an aborted computation. Therefore, if you are maintaining the rewrite path and abort some proof attempt (say, out of frustration over the time it is taking or because a stack overflow occurs) you can still inspect the path as it stood at the time of the abort. See **BREAK-REWRITE**.

Warning: It is dangerous to interrupt the theorem prover in the middle of a proof and call **MAINTAIN-REWRITE-PATH**. This can lead to hard errors later in the proof attempt because, when the path is being maintained assumptions are made about its structure and these assumptions are false if the path has not been maintained from the initial entry into the rewriter. If the theorem prover is running silently for long periods and you wish to poke around with **BREAK-REWRITE** to see what is going on, and the rewrite path is not enabled, you are advised to abort the proof, enable path maintenance, and then start the proof from scratch.

12.41. MAKE-LIB

General Forms:

```
(MAKE-LIB file)
```

and

```
(MAKE-LIB file compile-flg)
```

Example Form:

```
(MAKE-LIB "demo")
```

MAKE-LIB is the routine for saving the data base so that it can be restored by **NOTE-LIB**. The arguments are evaluated. The first argument, **file**, should be a valid file root name (see page 339). The second argument, if supplied, should be **T** or **NIL**; if not supplied, it defaults to **NIL**. The data base is saved to a pair of files, namely the **lib** and **lisp** extensions of **file** (see page 339).

Into the **lisp** extension of **file** **MAKE-LIB** writes the Lisp source code for the executable counterparts of all functions in the current data base. Into the **lib** extension it writes everything else in the data base. **MAKE-LIB** does not alter the data base. When the two files have been written, they are closed. If **compile-flg** is **T**, the Lisp source code for the executable counterparts is then compiled and the compiled code is then loaded. This will create a **bin** file extension corresponding to the just-created **lisp** file extension of **file**. **MAKE-LIB** returns a list containing the **lib** and **lisp** file names created.

12.42. META

META is the first element in one of the rule class tokens that may be given to **ADD-AXIOM**, **CONSTRAIN**, **FUNCTIONALLY-INSTANTIATE** or **PROVE-LEMMA** to indicate what kinds of rules should be generated from a formula. The general form of the token is **(META fn₁ ... fn_n)**. **META** rules are used in the simplification process, which is the first process tried. We describe the simplification process starting on page 266. Starting on page 289 we describe how **META** rules are generated and how they are used.

12.43. Names—Events, Functions, and Variables

In the formal treatment of the logic in Chapter 4 we defined the variable and function symbols of our logic as follows:

Terminology. A sequence of characters, *s*, is a *symbol* if and only if (a) *s* is nonempty, (b) each character in *s* is a member of the set

```
{A B C D E F G H I J K L M
 N O P Q R S T U V W X Y Z
 0 1 2 3 4 5 6 7 8 9
 $ ^ & * _ - + = ~ { } ? < >}
```

and, (c) the first character of *s* is a letter, i.e., in the set

```
{A B C D E F G H I J K L M
 N O P Q R S T U V W X Y Z}.
```

Terminology. The *variable symbols* and *function symbols* of our language are the symbols other than **T**, **F**, and **NIL**.

However, in the extended syntax we introduced the notion of “well-formed” *s*-expressions and their “translations” to formal terms. We then adopted the

policy that all terms displayed thereafter (and used in the implementation) would in fact be well-formed s-expressions standing for their translations.

The result of this is that there is a discrepancy between the set of legal names in the formal syntax and the set of names you can actually use in formulas. For example, **QUOTE** is a legal function name, but there is no way in the implementation to write down an application of the function symbol **QUOTE**, and an error is caused if you try to define **QUOTE**.

We therefore give here the definitions of the *legal names* as that concept is practically defined by the extended syntax.

The variable names permitted in the implementation are the symbols, excluding **T**, **F**, and **NIL** as above.

The function names and event names permitted in the implementation are the symbols, as defined above, excluding the following:

- **CASE**, **COND**, **F**, **LET**, **LIST**, **LIST***, **NIL**, **QUOTE**, and **T**, and
- any symbol, such as **CADDR**, which starts with a **C**, ends with an **R** and contains only **As** and **Ds** in between.

A function or event name is *old* (or *new*) iff it is (or is not) a citizen of the data base (see page 319).

12.44. NOTE-LIB

General Forms:

(NOTE-LIB file)

and

(NOTE-LIB file compile-flg)

Example Form:

(NOTE-LIB "demo")

NOTE-LIB is the routine for restoring a data base saved by **MAKE-LIB**. The arguments are evaluated. The first argument, **file**, should be a valid file root name. The second argument, **compile-flg**, should be **T** or **NIL**; it defaults to **NIL** if not supplied. The **lib** and **lisp** extensions of **file** must exist and should have been created by **MAKE-LIB** (see the discussion of File Names, page 339). If **compile-flg** is **T**, the **bin** extension of **file** must exist and should have been created by the same call of **MAKE-LIB** (with **compile-flg T**) that created the **lisp** extension of **file**.

NOTE-LIB destroys the data base extant when it is executed and configures the data base to be that which was extant when the corresponding **MAKE-LIB** for **file** was executed. If **compile-flg** is **T**, **NOTE-LIB** loads the com-

piled code for the executable counterparts of the functions in the incoming data base.

Because Nqthm changes (slowly) over time, it is sometimes the case that a library file created by one release of Nqthm is in a different format from those created by another release. This is called a *library incompatibility*. When it occurs, i.e., when **NOTE-LIB** detects that the incoming library is in the wrong format, a message to that effect is printed. However, **NOTE-LIB** attempts to load the library anyway. While you should not trust the resulting data base, it can often be used to recover a list of events (e.g., **EVENTS-SINCE**) from which a new library can be constructed from scratch.

NOTE-LIB returns a list containing the names of the two files read.

Uncompiled Nqthm library files are portable from one host to another. That is, the brand of Common Lisp in use during **MAKE-LIB** is irrelevant to **NOTE-LIB**. A similar remark holds about the host processor. Thus, for example, library files created on a Sparc workstation can be noted by a user running on a Mac II. Of course, the compiled file created when **MAKE-LIB** is given a second argument of **T** is generally not portable between Common Lisps or host processors. Thus, when noting a file from another system, do not give **NOTE-LIB** a second argument of **T**.

12.45. NQTHM Mode

NQTHM mode refers to the set of axioms added to the data base by **BOOT-STRAP**. See the discussion of **BOOT-STRAP** on page 303.

12.46. PPE

General Form:

(PPE name)

Example Form:

(PPE 'REVERSE)

PPE stands for “PrettyPrint Event.” The argument, **name**, is evaluated and should be a citizen of the data base. The command prints out the **EVENT-FORM** of **name**.

12.47. PROVE

General Form:

```
(PROVE term)
```

Example Form:

```
(PROVE '(PROPERP (REVERSE X)))
```

PROVE is the theorem prover. The argument, **term**, is evaluated by Lisp and should be a term. **PROVE** attempts to prove **term**; it returns **PROVED** if the proof succeeds and **NIL** if the proof fails. The output is written to the terminal. There is no provision for hints. **PROVE** does not alter the data base. To silence **PROVE** so that no output is produced, execute `(SETQ IO-FN #'(LAMBDA () NIL))`. That is, bind or set the Common Lisp variable **IO-FN** to a no-op function of no arguments. To restore normal printing, restore **IO-FN** to its initial value.

12.48. PROVEALL

General Forms:

```
(PROVEALL file events)
```

and

```
(PROVEALL file events compile-flg)
```

Example Form:

```
(PROVEALL "demo"
  '((DEFN REV1 (X A)
      (IF (NLISTP X)
          A
          (REV1 (CDR X)
                (CONS (CAR X) A))))
    (PROVE-LEMMA REV1-IS-REV (REWRITE)
      (EQUAL (REV1 X A)
              (APPEND (REVERSE X) A)))))
```

Note. At the time the first edition of this handbook was written, **PROVEALL** was the standard way to execute a sequence of event commands and collect the output into a file. It suffers however from the problem that its second argument, **events**, is a list of events and those events have to be created somehow, usually by reading them from a file. That creation process opens the possibility that Nqthm will somehow be corrupted. It also leaves open the possibility that

the syntax of the events as displayed in the permanent record, i.e., the file from which they were read, is not strictly in Nqthm's extended syntax. For example, the source file might employ user-defined or implementation-dependent Common Lisp read macros. From such a permanent record it is often impossible to reconstruct what was actually proved. For the second edition, therefore, we introduced **PROVE-FILE** (see page 356) which is similar to **PROVEALL** but which also takes responsibility for reading the events from a specified file, enforces the extended syntax, and also enforces restrictions intended to ensure that the resulting data base is "endorsed." We now prefer to use **PROVE-FILE** instead of **PROVEALL**. We have left **PROVEALL** in the system for the sake of compatibility.

The arguments to **PROVEALL** are evaluated. The first argument, **file**, should be a file root name (see page 339). Five or six extensions of **file** will be created, namely the **proofs**, **err**, **lib**, **lisp**, and, possibly, the **bin** and **fail** extensions. The second argument, **events**, should be a list of event forms to be executed. The third argument, **compile-flg**, should be **T** or **NIL** if supplied; if not supplied, it defaults to **NIL**.

PROVEALL first opens the **proofs** and **err** extensions of **file** and arranges for the event commands and **DO-EVENTS** to print all their "normal" output to the **proofs** extension and all error messages to the **err** extension. **PROVEALL** then prints a header into the **proofs** extension, giving the current time and date.

PROVEALL next uses **DO-EVENTS** to execute the event forms in **events**. **DO-EVENTS** executes each event form in turn until either they have all been executed or some form either causes an **ERROR** or **FATAL ERROR** or some **PROVE-LEMMA** fails. **DO-EVENTS** creates a complete transcript of the session in the **proofs** extension of **file**. In particular, the **proofs** extension will contain each event form executed, any **WARNING** or error messages caused, the justifications of each definition, the proof attempt for each **PROVE-LEMMA**, the time triple for each event, and the value of each event. The **err** extension will contain every **WARNING**, **ERROR**, and **FATAL ERROR** message printed in the **proofs** extension. Because the normal output of the event commands is being written to files, **DO-EVENTS** indicates its progress through **events** by printing just the event names to the terminal, each name being printed upon the commencement of the execution of the corresponding event.

Upon termination of **DO-EVENTS**, **PROVEALL** prints a "system description" into the **proofs** extension. This description gives the names of the files that were loaded to produce the version of the theorem prover used in the **proveall**. The format used differs from one host system to the next. **PROVEALL** also prints to the **proofs** extension the total amount of time consumed by the run, by summing (componentwise) the time triples printed.

PROVEALL then uses **MAKE-LIB** to save the data base, passing it the **compile-flg** argument. This creates both the **lib** and **lisp** extensions of **file**, and the **bin** extension if **compile-flg** is **T**. Note that if **DO-EVENTS** terminates prematurely because of an error or unsuccessful proof, the data base saved will be the one in which the terminal event was executed. The output from that event will be in the **proofs** extension. The failure can be reproduced by using **NOTE-LIB** to restore the data base created by **PROVEALL** and then executing the terminal event.

Finally, if **DO-EVENTS** terminated prematurely, **PROVEALL** creates the **fail** extension of **file** and writes into it (and into the **err** extension) an **ERROR** message containing the event form that caused the failure.

12.49. PROVE-FILE

General Forms:

```
(PROVE-FILE file)
```

and

```
(PROVE-FILE file :CHECK-SYNTAX      cs-flg
                  :WRITE-FLAG-FILES wf-flg
                  :PETITIO-PRINCIPII pp-flg)
```

Example Forms:

```
(PROVE-FILE "basic")
```

and

```
(PROVE-FILE "basic"
             :PETITIO-PRINCIPII T
             :WRITE-FLAG-FILES NIL)
```

PROVE-FILE is the standard way to create an “endorsed” data base from a file purporting to be a sequence of Nqthm event commands.⁴⁰ We define what we mean by an “endorsed” data base below, but the key property of such a data base is that we believe the formulas labeled “theorems” in it are actually theorems.

PROVE-FILE evaluates its arguments. The first argument, **file**, should be a file root name (see page 339), **file** must not end with the substring **"tmp"**, and **file.events**, the **events** extension of **file**, should exist and contain

⁴⁰We now prefer **PROVE-FILE** to the older **PROVEALL**, for the reasons given in the discussion of **PROVEALL**, page 354.

Nqthm event commands as described below. In accordance with Common Lisp, the optional “keyword arguments” may be supplied in any order or be omitted. When omitted, the **:CHECK-SYNTAX** and **:WRITE-FLAG-FILES** arguments default to **T**; **PETITIO-PRINCIPII** defaults to **NIL**.

PROVE-FILE is a read-eval-print loop similar to Common Lisp’s **LOAD** but enforcing certain requirements on the forms read and evaluated. **PROVE-FILE** repeatedly reads a form from **file.events**, prints the form, checks the syntax of the form, executes the form and prints its value. If all the forms in **file.events** are successfully processed, **PROVE-FILE** returns **T**. If an unacceptable form is read, **PROVE-FILE** calls **(MAKE-LIB file)** to save the data base and returns **NIL**.

By making the last form in **file.events** a **MAKE-LIB** form the user can arrange for successful termination to produce a library.

All output produced during **PROVE-FILE** is directed to Common Lisp’s ***STANDARD-OUTPUT***, which is normally the terminal. Because **PROVE-FILE** prints the forms and their values, the text written to ***STANDARD-OUTPUT*** resembles an interactive session. The user wishing to direct the output to a file should see **PROVE-FILE-OUT**.

PROVE-FILE observes a certain protocol of creating and deleting “flag files” to signal success or failure to a user who might be “watching” a **PROVE-FILE** in the background of some other activity. This protocol is explained below.

Each form in the **file.events** must be one of those described below. In addition, the first form must be a **BOOT-STRAP** or **NOTE-LIB** and only the last form is permitted to be a **MAKE-LIB**. A form, **form**, may occur in **file.events** only if:

- **form** is **(BOOT-STRAP)**, **(BOOT-STRAP NQTHM)** or **(BOOT-STRAP THM)**;
- **form** is **(NOTE-LIB str)** or **(NOTE-LIB str flg)**, where **str** is a string and **flg** is either **T** or **NIL**;
- **form** is a call of **ADD-AXIOM**, **ADD-SHELL**, **AXIOM**, **COMMENT**, **CONSTRAIN**, **DCL**, **DEFN**, **DEFTHEORY**, **DISABLE**, **DISABLE-THEORY**, **ENABLE**, **ENABLE-THEORY**, **FUNCTIONALLY-INSTANTIATE**, **LEMMA**, **PROVE-LEMMA**, **SET-STATUS**, **TOGGLE**, **TOGGLE-DEFINED-FUNCTIONS** or **UBT**;
- **form** is **(COMPILE-UNCOMPILED-DEFNS "tmp")**⁴¹;

⁴¹**PROVE-FILE** may not be used with files that end in **"tmp"** so as to avoid the possibility of a collision of the library file created for **file**, namely **file.lisp**, and the temporary file created by **COMPILE-UNCOMPILE-DEFNS**, **tmp.lisp**. Such a collision is possible, even when **file** is not **"tmp"** since **file** might contain directory prefixes.

- **form** is **(SETQ REDUCE-TERM-CLOCK n)**, where **n** is an integer;
- **form** is **(SETQ *COMPILE-FUNCTIONS-FLG* flg)**, where **flg** is either **T** or **NIL**; or
- **form** is **(MAKE-LIB str)** or **(MAKE-LIB str flg)**, where **str** is a string and is equal to the first argument of **PROVE-FILE**, **file**, and **flg** is either **T** or **NIL**.

All of the **event** files in our collection of Nqthm benchmarks in the **examples** directory (page 423) are processed successfully by **PROVE-FILE** and therefore illustrate the allowed format.

The readtable used while reading forms from **file.events** supports only the Nqthm extended syntax (page 138). Thus, some legal Common Lisp forms, such as **#3r102** (the integer eleven written in base three) and **'#, (lisp-fn)** (a **QUOTED** constant created by executing the Common Lisp program **lisp-fn**), are disallowed in **file.events**; errors will be signaled when such forms are read.

Caution. Because **PROVE-FILE** binds ***READTABLE*** to implement the Nqthm syntax, you may find it difficult to get out of an interactive break under **PROVE-FILE**. This is because our readtable prevents the use of keywords such as **:q** or **:a**, which are often chosen by implementors as the escape commands for interactive breaks. We try to restore the readtable appropriately before every error break, but circumstances beyond our control sometime prevent this. If an error occurs during **PROVE-FILE** and you are left in an interactive break and have difficulty (e.g., errors arise when you type a colon), we recommend that you execute **(SETQ *READTABLE* (COPY-READTABLE NIL))**, which will restore the host Lisp's readtable and allow you to type normally thereafter. The reason we prevent colons from occurring in Nqthm syntax is that the symbol **SI::CAR**, for example, might be **CAR** in some Lisps and not be **CAR** in others, depending on what symbols were imported into the **SI** package.

If the **:CHECK-SYNTAX** keyword argument is supplied and **NIL**, then we do not use the Nqthm readtable or enforce restrictions on the forms in the **file.events**.

A "flag file" protocol is observed by **PROVE-FILE** if the keyword argument **:WRITE-FLAG-FILES** is non-**NIL**, as it is by default. The protocol uses three extensions of **file**, **started**, **fail**, and **proved**. When called, **PROVE-FILE** creates **file.started** and deletes both **file.proved** and **file.fail** (when they exist). An error is caused if **file.proved** exists and cannot be deleted. Upon successful termination, **PROVE-FILE** creates **file.proved** and deletes **file.started**. Upon unsuccessful termination, **PROVE-FILE** creates **file.fail**.

Thus, barring interference from other users, a sure way to check for successful termination of a background execution of **PROVE-FILE** is to confirm that **file.proved** exists and was created after the **PROVE-FILE** commenced (since it is possible that the attempt to delete the file failed and left an error message on the terminal of the processor on which it was running). The existence of a **file.fail** created after the **PROVE-FILE** commenced is a sure way to confirm the failure of **PROVE-FILE**. But failures may manifest themselves by nontermination or resource exhaustion that are not detectable with the flag file protocol.

When **file.proved** is created, certain interesting statistics are written into it. These include the total time used in processing **file.events**, a note on the incompatibility of the library files noted if such an incompatibility has occurred in the session (see **NOTE-LIB**), the names of all nondefinitional axioms involved (including those in library files noted), and the names of all events targeted by the undo-back-through command **UBT** during the run.

The **:PETITIO-PRINCIPII** argument, when non-**NIL**, makes **PROVE-FILE** skip all proof obligations. Thus, with this argument set, the forms in **file.events** are merely *assumed* to be admissible (though their syntax is checked). This is a fast way to almost reconstruct the data base created by a sequence of events. The reconstruction is not perfect because the logical dependencies discovered during proofs are not recorded. It is a handy way to experiment or to recover from system crashes. However,

Warning. Nqthm is unsound when the **:PETITIO-PRINCIPII** argument to **PROVE-FILE** is used. No record of the use of **:PETITIO-PRINCIPII** is left in the data base or in libraries created by the user from the data base constructed by **PROVE-FILE**.

When **PETITIO-PRINCIPII** is non-**NIL**, **PROVE-FILE** does not observe the flag file protocol, and hence, does not create **file.proved** even if the run is successful. Furthermore, no **MAKE-LIB** is done by such a **PROVE-FILE** and no output occurs. The routine **SKIM-FILE** is a convenient way to call **PROVE-FILE** with **PETITIO-PRINCIPII** set.

We conclude with some motivational remarks about **PROVE-FILE** and the process of creating endorsed data bases.

PROVE-FILE is a weak, file-based read-eval-print loop. Why not stick with the all-powerful Common Lisp read-eval-print loop and its file-based counterpart, **load**?

During the formalization and proof development phase of an Nqthm project, we prefer the flexibility and sense of freedom one gets from dealing directly with Lisp, rather than with some pea-brained and constraining “user interface.” For example, we often find ourselves executing raw Common Lisp forms to “compute” the Nqthm forms we wish to evaluate. We enjoy being able to type arbitrary Common Lisp forms, define Lisp utilities and readmacros, and other-

wise tailor the environment to our particular tastes. We thus recommend against the casual use of **PROVE-FILE** in the development phase of a project.

But the Common Lisp interface is “too powerful” when it comes to soundness. Executing (**SETQ TRUE FALSE**) in the host Lisp will render Nqthm unsound. More subtle forms of unsoundness can creep in, e.g., by the accidental corruption of Nqthm’s data base caused by destructive list processing operations performed by utility routines written by the user. As noted on page 390, one of Nqthm’s own utilities, **UNDO-NAME**, can render the system unsound. Therefore, we do not endorse as a theorem anything “proved” by Nqthm in a normal interactive session. When a proving project is entirely complete, we prefer to know that no idiotic, surreptitious, or untrusted Lisp code is present that would render our efforts meaningless. Thus it makes sense to have, as we do in **PROVE-FILE**, a loader that prohibits the intermingling of possibly dangerous Lisp forms with trusted Nqthm forms.

When we have reached the end of a project we have often constructed a single **events** file, say **project.events**, that starts with a **BOOT-STRAP**. If we wish to preserve the final data base of the project, as we would if other projects build upon it, we insert a (**MAKE-LIB "project"**) at the end of the file. We then undertake to “certify” the event list. We start by checking that there is no file named **project.proved** on the directory. Then we invoke (**PROVE-FILE "project"**). If this terminates with the creation of **project.proved** we inspect its contents to see that no inadvertent axioms were added.⁴² We also look for **UBTs** in the script because the presence of undoing in an event list can lead to such confusing results as a single name being defined two different ways (at different times).

Because projects that use Nqthm can be very large, taking years to complete, and can be worked on by many people, there is often the practical necessity to divide the events into a collection of several files which are chained together by **NOTE-LIBs** and **MAKE-LIBs**. To ‘certify root and branch’ a collection of such files using **PROVE-FILE**, we start by making sure that there are no **lib**, **lisp**, **bin**, **proved**, or **started** file extensions on the directories in question, including those referred to in the **NOTE-LIB** forms that occur at the beginnings of the files to be certified. (Alternatively, we check that every file “in sight” has extension **events**.) This check will prevent the use of older libraries and thus avoid the risk of using libraries that are uncertified or otherwise out of date with

⁴²One can develop an Nqthm proof in a top-down fashion by using **ADD-AXIOM** to state the lemmas one intends later to prove. One then proceeds to derive the main result from these unproven “lemmas.” Upon completing the main proof, one is then obligated to replace each of the **ADD-AXIOMs** with a suitable event list concluding in a **PROVE-LEMMA**. Occasionally an **ADD-AXIOM** is forgotten or overlooked and finds its way into what the user thought was the completed script.

respect to their source files. Then we execute a sequence of **PROVE-FILE** commands on the event files in question in an order consistent with the inner dependencies (calling **PROVE-FILE** on each file exactly once). If at the end there is a **proved** file for each of the **event** files, the certification is complete. Obviously, there should be no changes permitted to any of the files in sight (except, of course, the creation of new files by **PROVE-FILE**) during the execution of the **PROVE-FILES**.

The collection of example event files distributed with Nqthm illustrates this discipline. We recommend these examples to the reader. To maintain this collection we have mechanized the certification procedure with a Lisp script, distributed as **driver.lisp** in the **examples** subdirectory.

Technically speaking, an *endorsed data base* is a data base created, in a fresh Nqthm, by a successful execution of **PROVE-FILE**, provided (a) the first event in the **events** file is either a **BOOT-STRAP** or a **NOTE-LIB** of an endorsed data base, and (b) no ‘library incompatibilities’ were noted (see **NOTE-LIB**). Modulo the difficulties discussed below, we believe the ‘theorems’ in an endorsed data base are indeed theorems. In particular, consider any (**PROVE-LEMMA name rule-classes term hints**) in the endorsed data base. Let *h* be the sequence of axiomatic acts in the data base prior to the event in question. Then we claim *h* is a history (i.e., the foregoing axiomatic acts are all admissible) and **term** can be proved directly from the axioms of some definitional/constrain extension of *h*.

Inadequacies. We currently know of three senses in which **PROVE-FILE** fails to do fully the job that we intend. (1) Common Lisp syntax for integers is not entirely fixed by the Common Lisp standards. For example, **1.7J** is given as an explicit example, on p. 341 of [131], of a character sequence that is a ‘potential number,’ that is, a sequence that some Common Lisp might read as a number. If a Common Lisp were to read **1.7J** as an integer, e.g., as the integer 17, then by running Nqthm in that Common Lisp one could prove the theorem (**EQUAL 1.7J 17**). But (**EQUAL 1.7J 17**) might be false, or might even cause an error in Nqthm in another Common Lisp. The solution to this problem is to check with your Lisp provider that no character strings that are ‘potential numbers’ read as integers, except those satisfying the normal syntax for integers shared by all Common Lisp implementations. (2) It is not clear from the Common Lisp standard what files can be created by the compilation process. Typically, if you compile **project.lisp**, then **project.x** is created, for some extension *x*. However, GCL takes this freedom to a rather liberal extent: when the variable **COMPILER:*SPLIT-FILES*** is non-**NIL**, as we have been forced to set it when doing the **fm9001-piton** proofs, files with names **0project.o**, **1project.o**, **2project.o**, ..., may be created during the compilation of **project.lisp**. Such names create a potential for collision with the compiled library of another events file, say one named

0project.events. Thus the GCL user of **PROVE-FILE** should check that the name of no events file begins with a digit. We do not make this check mechanically, as we do the check for names that end in **"tmp"**, because it is possible that some prefix of a name supplied to **PROVE-FILE** is a mere directory name, e.g., **2nd-try/project**. (3) Unbalanced close parentheses at the top level are treated by Common Lisp's **READ** in an undefined, implementation dependent way. Most Lisps we use ignore unbalanced close parentheses, but some cause an error. Such parentheses are "invalid" according to [131]. A perfect implementation of **PROVE-FILE**, we think, would check for such parentheses and cause an error.

12.50. PROVE-FILE-OUT

General Forms:

```
(PROVE-FILE-OUT file)
```

and

```
(PROVE-FILE-OUT file
                  :CHECK-SYNTAX      cs-flg
                  :WRITE-FLAG-FILES wf-flg
                  :PETITIO-PRINCIPII pp-flg)
```

Example Forms:

```
(PROVE-FILE-OUT "basic")
```

and

```
(PROVE-FILE-OUT "basic"
                  :PETITIO-PRINCIPII T
                  :WRITE-FLAG-FILES NIL)
```

PROVE-FILE-OUT is like **PROVE-FILE** except it arranges for all output to go to the **proofs** extension of **file**. That is, **PROVE-FILE-OUT** opens the file **file.proofs** and calls **PROVE-FILE** (on the same arguments given to **PROVE-FILE-OUT**) in such a way as to ensure that all output is written to **file.proofs**. The output file is closed upon termination and represents a complete transcript of the processing of **file.events**. Nothing is printed to the terminal. See **PROVE-FILE**.

12.51. PROVE-LEMMA

General Forms:

```
(PROVE-LEMMA name rule-classes term)
and
(PROVE-LEMMA name rule-classes term hints)
```

Example Form:

```
(PROVE-LEMMA REVERSE-REVERSE (REWRITE)
  (IMPLIES (PROPER X)
    (EQUAL (REVERSE (REVERSE X)) X)))
and
(PROVE-LEMMA TIMES-LIST-EQUAL-FACT (REWRITE)
  (IMPLIES (PERM (POSITIVES N) L)
    (EQUAL (TIMES-LIST L) (FACT N))))

;Hints:
((USE (PERM-TIMES-LIST (L1 (POSITIVES N))
  (L2 L)))
  (DISABLE PERM-TIMES-LIST)))
```

PROVE-LEMMA is an event command for proving a theorem and storing it and possibly some generated rules in the data base. It does not evaluate its arguments. **name** must be a new name and will be made the name of the event in the data base, **rule-classes** must be a (possibly empty) list of rule classes and **term** must be a well-formed term (see **TRANSLATE**). **PROVE-LEMMA** applies the theorem prover to **term**, printing commentary as the proof attempt proceeds. If the proof attempt is successful, **term** is stored in the data base as a theorem; in addition, rules of the classes specified by **rule-classes** are generated from **term** and stored. The name of each rule is **name**. An error is caused if **term** is unsuitable for some class in **rule-classes** and no change is made in the data base.

When formulating input for **PROVE-LEMMA** you must be cognizant of the rule interpretation of **term** in addition to its mathematical content. See Chapter 11 for a detailed explanation of the effect of each type of rule and how rules are generated from formulas and rule classes. See Chapter 13 for advice about how to use rules.

Note that if **rule-classes** is **NIL** then the term is stored as a theorem and will be available for **USE** hints (see below) but will generate no rules.

If the proof attempt succeeds, **PROVE-LEMMA** prints “**Q.E.D.**” and a time triple for the event and returns the name of the new event, **name**.

If the proof attempt fails, **PROVE-LEMMA** prints a “failure banner”

```
***** F A I L E D *****
```

and then a time triple. Then it returns **NIL**.

When the theorem prover fails the only claim made is the trivial one that our heuristics did not lead to a proof. The system’s remarks should not be interpreted to mean that the input formula is not a theorem, even if the last formula printed is **F**: the heuristics might have generalized the input formula before failing. That said, however, two caveats are in order.

First, conjectures submitted for proof are frequently not theorems. Do not be too quick to dismiss a failed proof simply because the theorem prover is so weak. It is indeed weak, but you can’t fool it either. If a point were awarded the opposition each time a player asserted an inconsistent belief, the theorem prover would never lose and would, in fact, win almost every session.

Second, when it fails on *theorems* it is frequently because the proof you had in mind involves some step that has not been made explicit by your previously proved lemmas—or at least by their interpretation as rules. See Chapter 9 where we discussed the notion of the crucial “check points” in a proof attempt and what you should do at each. If worse comes to worse, you can essentially tell the system your proof via the **hint** argument to **PROVE-LEMMA**, which we describe at length below.

The **hints** argument permits the user to give the theorem prover some guidance in its proof attempt. If nonempty, **hints** must be a list of tuples. The **CAR** of each tuple must be one of the symbols listed below, where we describe the form and effect of each acceptable tuple. An error is caused if any element of **hints** fails to have the correct form or if there is more than one occurrence of any kind of hint.

12.51.1. *USE Hints*

General Form:

```
(USE (name1 (v1,1 t1,1) ... (v1,n1 t1,n1))
...
(namek (vk,1 tk,1) ... (vk,nk tk,nk)))
```

Example Form:

```
(USE (LEMMA1 (X (CAR A)) (Y 0))
(LEMMA2 (X (SUB1 X)) (I X) (K (FACT X))))
```

A **USE** hint essentially forces the use of one or more instances of one or more previously proved theorems, definitions, or axioms. Each pair following the

symbol **USE** specifies the name of a formula and a substitution. The **USE** hint creates the indicated instantiation of each formula and adds it as an explicit hypothesis to the conjecture, **term**, submitted to the theorem prover.

More precisely, the conditions on **USE** hints are as follows. Each **name_i** must be a citizen of the data base with a non-**NIL** associated **FORMULA** in **DATA-BASE**. It does not matter if **name_i** is enabled or disabled or if any rules were generated for **name_i**. Each **v_{i,j}** must be a variable symbol, each **t_{i,j}** must be a term, and **v_{i,j}** must be distinct from **v_{i,k}** if **j ≠ k**.

Provided the above conditions are met, the effect of such a **USE** hint on the behavior of **PROVE-LEMMA** is to cause a modified (but provably equivalent) conjecture to be submitted to the theorem prover. Rather than submitting **term**, **PROVE-LEMMA** submits

(**IMPLIES** (**AND** **thm₁** ... **thm_k**) **term**)

where **thm_i** is obtained by instantiating the **FORMULA** for **name_i**, replacing **v_{i,j}** by **t_{i,j}**.

Observe that each **thm_i** is provably non-**F**—each is an instance of a theorem, definition, or axiom. Hence, the submitted formula is equivalent to the input one.

However, the careful reader might also note that if **name_i** is enabled and is a well-constructed replacement rule, there is a very high likelihood that it will be used to eliminate **thm_i** from the modified conjecture. This renders the hint useless. It is generally a good idea to **DISABLE** any **REWRITE** rules instantiated with a **USE** hint.

The modified conjecture is often easier to prove because the necessary instances of the necessary theorems are explicitly present. Indeed, it is possible to reduce any noninductive proof in the logic to propositional calculus and equality reasoning after supplying all the necessary instances of lemmas.⁴³

12.51.2. *EXPAND Hints*

General Form:

(**EXPAND** **t₁** ... **t_n**)

Example Form:

(**EXPAND** (**PRIME1** **X** (**SUB1** **X**)))

⁴³We do not recommend such a wholesale use of **USE** simply because the number of lemmas and instances needed would often produce a modified conjecture that was too large for the combinatorial processing done by all known propositional decision engines.

Each t_i must be a term of the form $(fn\ x_1 \dots x_k)$ where fn is a defined function. That is, each t_i must be a call of a defined function. Variable symbols, calls of shell functions, and calls of undefined or constrained functions are prohibited. The **EXPAND** hint forces the expansion of each t_i whenever it is encountered by the rewriter.

The **EXPAND** hint is implemented simply by putting the (translated and with certain calls of **IDENTITY** expanded, as described below) t_i on a list that is known to the rewriter. Whenever a function call is considered for expansion the list is checked. Thus, it is not necessary that the t_i occur in the input conjecture. However, it is crucial that the *exact* form of the intended term be written in the hint. We give an example below.

Suppose $(\text{SECOND } X)$ is defined to be $(\text{CADR } X)$. Suppose also that $(\text{TIMES } I (\text{SECOND } X))$ occurs in the input conjecture and the intended proof requires its expansion but the system does not expand it automatically. An **EXPAND** hint is called for. The hint $(\text{EXPAND } (\text{TIMES } I (\text{SECOND } X)))$ will *not* work. By the time the rewriter considers expanding the **TIMES** expression, it will have become $(\text{TIMES } I (\text{CADR } X))$. Thus, to force the expansion of $(\text{TIMES } I (\text{SECOND } X))$ it is necessary to give the hint $(\text{EXPAND } (\text{TIMES } I (\text{CADR } X)))$! This is not generally a problem. The reason is that you will generally add **EXPAND** hints only in response to the failure of an earlier proof attempt. In that earlier attempt the term $(\text{TIMES } I (\text{CADR } X))$ will have appeared at the checkpoint where you expected its expansion to appear and you will have realized “the theorem prover needs to expand *that* term.”

As with the left-hand sides of **REWRITE** rules, it is sometimes desired to write large constants in the terms to be expanded. An alternative is to write a variable-free expression that reduces to the desired constant and to embed that expression in a call to the **IDENTITY** function. The **EXPAND** hint replaces such expressions by their reduced values before storing them on the list inspected by the rewriter. Thus, when used in an **EXPAND** hint, $(\text{SUM } A\ B\ 4294967296)$ and $(\text{SUM } A\ B\ (\text{IDENTITY } (\text{EXPT } 2\ 32)))$ cause identical behavior, provided **EXPT** is defined as the exponentiation function. The **IDENTITY** function is given special treatment only here, in the **EXPAND** hint, and in the preprocessing of **REWRITE** rules (see page 277). Otherwise, **IDENTITY** is not given any special treatment by the system.

12.51.3. *DISABLE and ENABLE Hints*

General Forms:

(DISABLE name₁ ... name_n)

and

(ENABLE name₁ ... name_n)

Example Form:

(DISABLE LEMMA1 TIMES-2-REWRITE)

The **DISABLE** hint has the effect of disabling the listed rule names for the duration of the proof attempt in question. The **ENABLE** hint is analogous.

We sometimes call these hints “local” enables and disables because they do not affect the status of the names in the data base. The effects of these hints are visible only during the proof attempt for which the hints were supplied.

Each **name_i** must be a citizen of the data base. See page 328 for a discussion of the effect of enabling or disabling a name and why it is done. Roughly speaking, the effect of **DISABLE** is to prevent the theorem prover from using any of the named rules during the current proof attempt. **ENABLE** permits the named rules to be used even though they might be **DISABLED** in the current data base.

Using the **DISABLE-THEORY** and **ENABLE-THEORY** hints, all the names in a theory can be “locally” disabled or enabled. This raises questions of priority, e.g., “What if a name is **DISABLED** locally with a **DISABLE** hint but is a member of a theory that is enabled locally?” Such questions are addressed in the following discussion of the **DISABLE-THEORY** and **ENABLE-THEORY** hints.

12.51.4. *DISABLE-THEORY and ENABLE-THEORY Hints*

General Forms:

(DISABLE-THEORY theory-name₁ ... theory-name_n)

and

(ENABLE-THEORY theory-name₁ ... theory-name_n)

Example Form:

(DISABLE-THEORY LIST-THEORY BAG-THEORY)

The **DISABLE-THEORY** hint has the effect of disabling all the names in the listed theories. The **ENABLE-THEORY** hint is analogous.

We sometimes call these hints “local” enables and disables because they do not affect the status of the names in the data base. The effects of these hints are visible only during the proof attempt for which the hints were supplied.

Each **theory-name_i** must be a theory name, i.e., a name introduced with **DEFTHEORY** or one of the two built-in theory names, **GROUND-ZERO** and **T**. See page 328 for a discussion of the effect of enabling or disabling a name and why it is done.

If the theory name **T** is used then no other theory name is permitted in the hint. Furthermore, if the hint **(DISABLE-THEORY T)** is supplied, then **(ENABLE-THEORY T)** may not be supplied and vice versa.

The local status of a name during a proof may thus be affected by any of four hints, **DISABLE**, **ENABLE**, **DISABLE-THEORY**, and **ENABLE-THEORY**. The following algorithm is used to determine the status of a name during a proof attempt governed by any of these hints.

- If the given name is included in an **ENABLE** hint, consider it enabled.
- Otherwise, if the given name is included in a **DISABLE** hint, consider it disabled.
- Otherwise, if the given name belongs to some theory named in an **ENABLE-THEORY** hint, consider it enabled.
- Otherwise, if the given name belongs to some theory named in an **DISABLE-THEORY** hint, consider it disabled.
- Otherwise, if there is an **(ENABLE-THEORY T)** hint (respectively, a **(DISABLE-THEORY T)** hint), consider it enabled (respectively, disabled).
- Otherwise, simply check whether it is enabled or disabled in the global database.

Thus, roughly speaking, the hints which explicitly mention rule names, **DISABLE** and **ENABLE**, are given priority over the theory-level hints, **DISABLE-THEORY** and **ENABLE-THEORY**; enabling is given priority over disabling. For example, one may give a hint that enables several relevant theories but then one may explicitly disable certain names within those theories.

12.51.5. HANDS-OFF Hints

General Form:

(HANDS-OFF $fn_1 \dots fn_n$)

Example Form:

(HANDS-OFF PLUS TIMES QUOTIENT REMAINDER)

The **HANDS-OFF** hint prevents the rewriter from trying any rule that rewrites a term beginning with one of the function symbols listed in the hint. Each fn_i must be a function symbol. Each such name is added to a list known to the rewriter. Roughly speaking, if a function symbol is on this list then every **REWRITE** stored under that symbol, including its definition, is locally disabled. Because the documentation of the **HANDS-OFF** feature was grossly incorrect in the first edition of this handbook, we describe it somewhat more carefully.

When rewriting the target term ($fn\ a_1 \dots a_k$), the rewriter first rewrites each a_i , say to a_i' . The rewriter then shifts its attention to ($fn\ a_1' \dots a_k'$), which we here call the “intermediate term.” If all the a_i' are explicit values and fn has an enabled executable counterpart, the rewriter returns the explicit value obtained by reducing the intermediate term. If fn is **EQUAL** or a shell recognizer such as **NUMBERP**, built in rules are tried on the intermediate term. If the intermediate term is one of several special forms having to do with **V&C\$**, certain built-in rules are tried. See the Reference Guide entries for **REWRITE-APPLY\$**, **REWRITE-APPLY-SUBR**, **REWRITE-CAR-V&C\$**, **REWRITE-CAR-V&C-APPLY\$**, **REWRITE-EVAL\$**, **REWRITE-V&C\$**, and **REWRITE-V&C-APPLY\$**, starting on page 378. Otherwise, the rewriter determines whether fn is among the fn_i noted in the **HANDS-OFF** hint. If so, the intermediate term is returned as the result of rewriting the target term. Otherwise, the rewriter tries to apply enabled **REWRITE** rules matching the intermediate term, including the definition of fn .

12.51.6. INDUCT Hints

General Form:

(INDUCT ($fn\ v_1 \dots v_n$))

Example Form:

(INDUCT (PRIME1 X Y))

The **INDUCT** hint forces the theorem prover immediately into the induction process. Furthermore, rather than select an induction using its heuristics, the induction used is that suggested by ($fn\ v_1 \dots v_n$). fn must be a recur-

sively defined function, and the \mathbf{v}_i must be distinct variables. The induction suggested has exactly the same case analysis as in the body of the definition of **fn**. Those branches not leading to recursive calls are base cases. Those branches leading to $k > 0$ recursive calls $(\mathbf{fn} \ t_{1,1} \ \dots \ t_{1,n}), \dots, (\mathbf{fn} \ t_{k,1} \ \dots \ t_{k,n})$ have k induction hypotheses. The i^{th} induction hypothesis is obtained by instantiating each \mathbf{v}_j by the corresponding term $t_{i,j}$ from the i^{th} recursive call.

The output produced by the theorem prover in response to an **INDUCT** hint is very unenlightening: it prints nothing at all but simply produces the various cases as described above and proceeds to simplify each.

12.51.7. *DO-NOT-INDUCT*

General Form and Example Form:

(DO-NOT-INDUCT T)

If present, the **DO-NOT-INDUCT T** hint causes the theorem prover to abort with failure as soon as it encounters a goal for which it would otherwise enter the induction process. (The second element of the hint can, in fact, be any Lisp object and is ignored.)

12.51.8. *DO-NOT-GENERALIZE*

General Form and Example Form:

(DO-NOT-GENERALIZE T)

If present, the **DO-NOT-GENERALIZE T** hint disables the entire generalization process. (The second element of the hint can, in fact, be any Lisp object and is used as the value of an internal flag. If the flag setting is **NIL** the hint has no effect at all. Thus, the “General Form” is also the “Example Form.”)

12.51.9. *NO-BUILT-IN-ARITH*

General Form and Example Form:

(NO-BUILT-IN-ARITH T)

If present, the **(NO-BUILT-IN-ARITH T)** hint prevents the built-in linear arithmetic procedure from being used. We use this hint only to demonstrate the ability (or inability) of the system to prove by heuristic means the elementary

theorems about **LESSP** and **PLUS**. (The second element of the hint can, in fact, be any Lisp object and is used as the value of an internal flag. If the flag setting is **NIL** the hint has no effect at all. Thus, the “General Form” is also the “Example Form.”)

12.52. R-LOOP

General and Example Form:
(**R-LOOP**)

R-LOOP permits the evaluation of terms in the logic. It is a Lisp style “read-eval-print loop” providing certain convenient nonlogical features through so-called “special forms.” (Noninteractive entrances to **R-LOOP** are provided via the functions **R** and **S** described at the end of this section.)

Upon entry to **R-LOOP** the system prints a header indicating the values of certain “switch settings” explained below. Then **R-LOOP** prompts the user for type-in by printing a “*” and then waiting until an s-expression, **t**, has been typed. As usual in our system, one is actually typing to the standard Lisp reader. Line editing, redisplay of previously typed input, etc., are all available as however provided by the host Lisp system. In some Lisp systems, the user must type a carriage return after the s-expression before the system actually reads it; in other systems, the final balancing close parenthesis initiates the read.

Once **t** has been read it is evaluated under the axioms and definitions as described below, unless **t** is one of the special forms also noted below. The process of evaluation will yield one of three possible outcomes: an error message indicating that **t** is not well formed, a message indicating that **t** cannot be reduced to an explicit value, or an explicit value equivalent to **t**. In the latter case the value is printed. In all cases, **R-LOOP** then iterates by prompting for another s-expression.

Here is a sample exchange:

```
* (APPEND '(1 2 3) '(4 5 6))    ;user type-in
'(1 2 3 4 5 6)                 ;R-LOOP's response
```

Users familiar with the Lisp “read-eval-print” loop will notice a slight but disconcerting difference: in Lisp the exchange would have been

```
* (APPEND '(1 2 3) '(4 5 6))    ;user type-in
(1 2 3 4 5 6)                   ;Lisp's response
```

Note the absence of the **'** in Lisp’s response. **R-LOOP** deals exclusively with *terms*—the “values” printed are terms, they just happen to all be explicit values

and thus, when printed in our syntax, usually are preceded by the ' mark. (Numbers, **T**, **F**, and **NIL** are the only explicit values displayed without use of **QUOTE** notation.)

By "evaluation" we mean the reduction of a term to an explicit value by a call-by-value style interpretation of the axioms and definitions. For example, to evaluate a term of the form (**PERM** **x** **y**), **R-LOOP** recursively reduces **x** and **y** to explicit values and then (effectively) instantiates the body of **PERM** with those values and reduces the result.⁴⁴

The evaluation performed by **R-LOOP** takes place under an assignment of explicit values to variable symbols. **R-LOOP** permits the user to set this assignment with the **SETQ** special form. This feature is provided so that the user can save the value of one evaluation for possible input to another. Thus, if **x** has the value **7** and **y** has the value **8** then the evaluation of (**TIMES** **x** **y**) is **56**.

If the term **t** is submitted for evaluation and produces the explicit value **v**, then it is a theorem that (**EQUAL** **t** **v**), under the equality hypotheses implicit in the current assignment of values to variables.

Not all terms can be reduced to explicit values. Examples are variables not assigned values in the current **R-LOOP** assignment (i.e., "unbound variables" in the programmer's sense, "undefined abbreviations" in the mathematician's) and calls of undefined or constrained functions. Some terms which are provably equal to explicit values cannot be reduced to that explicit value by evaluation. For example, (**IF** **v** **3** **3**) is provably equal to **3** but does not reduce to **3** if **v** is "unbound." Similarly, (**V&C\$** **T** '(**RUSSELL**) **NIL**) is provably equal to **F**, if **RUSSELL** is defined as on page 95, but will not reduce to **F**. When a term submitted for evaluation cannot be reduced to an explicit value by **R-LOOP**, the answer "(**NOT REDUCIBLE**)" is printed.

R-LOOP has two mode switches that can be set by several special forms. One of the switches determines whether evaluation is "traced" and, if so, how. The other determines whether abbreviations are used when values are printed out.

The *trace mode* may be *full*, *partial*, or *off*. When trace mode is off, no printing occurs during the evaluation of terms. The value is printed when the evaluation has completed successfully. When trace mode is not off, printing occurs during the evaluation of each term. Trace mode prints a step-by-step proof of the equivalence of the input term and the derived value. The difference between full and partial trace mode is only how large the printed steps are. In full trace mode, every application of substitutions of equals for equals into the input term is printed. In partial trace mode, we print only the "major" steps associated with function expansion.

⁴⁴The "R" in "R-LOOP" in fact stands for "reduce" as we defined it in [25].

Suppose we define **APP** as **APPEND**:

Definition.

```
(APP X Y)
=
(IF (NLISTP X)
    Y
    (CONS (CAR X)
          (APP (CDR X) Y)))
```

Below we illustrate the evaluation of `(APP '(A) '(1 2 3))`, first with trace mode off, then in full trace mode, and then in partial trace mode. Then, we illustrate `(APP '(A B C) '(1 2 3))` in partial trace mode.

```
(R-LOOP)
Trace Mode: Off    Abbreviated Output Mode: On
Type ? for help.
*(APP '(A) '(1 2 3))
'(A 1 2 3)
*FULL-TRACE
Trace Mode: Full
*(APP '(A) '(1 2 3))
=(IF (NLISTP '(A))
      '(1 2 3)
      (CONS (CAR '(A))
            (APP (CDR '(A)) '(1 2 3))))
=(IF F
      '(1 2 3)
      (CONS (CAR '(A))
            (APP (CDR '(A)) '(1 2 3))))
=(CONS (CAR '(A))
        (APP (CDR '(A)) '(1 2 3)))
=(CONS 'A (APP (CDR '(A)) '(1 2 3)))
=(CONS 'A (APP NIL '(1 2 3)))
=(CONS 'A
        (IF (NLISTP NIL)
              '(1 2 3)
              (CONS (CAR NIL)
                    (APP (CDR NIL) '(1 2 3)))))
=(CONS 'A
        (IF T
              '(1 2 3)
              (CONS (CAR NIL)
                    (APP (CDR NIL) '(1 2 3)))))
='(A 1 2 3)
```



```

*TRACE
Trace Mode:  Partial

*(APP '(A) '(1 2 3))
=(CONS 'A (APP NIL '(1 2 3)))
='(A 1 2 3)

*(APP '(A B C) '(1 2 3))
=(CONS 'A (APP '(B C) '(1 2 3)))
=(CONS 'A
      (CONS 'B (APP '(C) '(1 2 3))))
=(CONS 'A
      (CONS 'B
            (CONS 'C (APP NIL '(1 2 3)))))
='(A B C 1 2 3)

*OK
Exiting R-LOOP.
NIL

```

The *output abbreviation mode* may be either *on* or *off*. The mode determines how the final explicit value of a term is displayed. When output abbreviation is on, explicit values are displayed in a way that does not require use of the ugly “*1*” prefix. When output abbreviation mode is off, explicit values are displayed in *QUOTE* notation. For example,

```

(R-LOOP)
Trace Mode: Off   Abbreviated Output Mode: Off
Type ? for help.

*(PAIRLIST '(A B C) (LIST 1 2 3))
'((A . 1) (B . 2) (C . 3))
*(PAIRLIST '(A B C) (LIST 0 T F))
'((A . 0)
  (B . *1*TRUE)
  (C . *1*FALSE))

*ABBREV
Abbreviated Output Mode:  On
*(PAIRLIST '(A B C) (LIST 1 2 3))
'((A . 1) (B . 2) (C . 3))
*(PAIRLIST '(A B C) (LIST 0 T F))

```

```
(LIST '(A . 0)
      (CONS 'B T)
      (CONS 'C F))
```

```
*OK
Exiting R-LOOP.
NIL
```

In the session above we start with abbreviated output mode off. We evaluate two **PAIRLISTS**. The values of both are printed in **QUOTE** notation. But the value of the second one requires the use of ***1*TRUE** and ***1*FALSE**.

Then we enter abbreviated output mode and re-evaluate both **PAIRLISTS**. The value of the first one is printed just as before. But the value of the second one is not printed in **QUOTE** notation. Instead, **LIST** and **CONS** are used to “unquote” enough of the value to expose those parts that would otherwise require use of the “***1***” prefix.

Below we list the special forms recognized by **R-LOOP**. The special forms are, in general, well-formed terms that are simply not treated as terms when read at the top level by **R-LOOP**.

OK Exit **R-LOOP** and return to its caller. If you invoked (**R-LOOP**) from the Lisp command interpreter (as opposed to from within some other Lisp program), **OK** returns you to the Lisp command interpreter. No harm results from exiting **R-LOOP** via Lisp interrupt characters (see “Aborting Commands” page 295). Variable assignments in the **R-LOOP** assignment are maintained from one call of **R-LOOP** to the next.

(**SETQ** *var term*) In any s-expression of this form, *var* should be a variable symbol and *term* should be a term. This special form causes *term* to be evaluated under the current **R-LOOP** assignment and, if the evaluation succeeds and produces *val*, the current assignment is changed so that *var* has the value *val*. In addition, *val* is printed. If the evaluation of *term* fails, no change is made to the **R-LOOP** assignment.

TRACE Set trace mode to “partial.”

FULL-TRACE Set trace mode to “full.”

UNTRACE Set trace mode to “off.”

ABBREV Turn on abbreviated output mode.

UNABBREV Turn off abbreviated output mode.

- ? Causes **R-LOOP** to print out a brief summary of the available special forms.

Special forms are given special treatment only when they are the top-level s-expression read by **R-LOOP**. If, for example, the **SETQ** special form is used as a proper subterm in an s-expression, it is evaluated in the usual sense of the logic. For example, suppose **X** is **7** in the **R-LOOP** assignment, and the user types the s-expression **(PLUS (SETQ X 3) X)** to **R-LOOP**. Unless the user has defined the function **SETQ** in the logic, this s-expression is ill-formed. If **SETQ** has been defined, the value of this s-expression is the value of **(PLUS (SETQ 7 3) 7)**.

The **R-LOOP** assignment is maintained from one call of **R-LOOP** to the next within a given session with the theorem prover. A typical use of **R-LOOP** in a session is to enter it periodically after function definitions to test the suitability of the definitions or the validity of a conjecture being formulated. The variable assignment feature is often used to store test data, e.g., the state argument of an interpreter for a programming language being formalized.

The variable assignment is not written to the library files created by **MAKE-LIB**. Unless otherwise saved, **R-LOOP** assignments are lost when the Lisp session terminates. The assignment is stored in an internal form as the value of the Lisp variable **R-ALIST**. This value may be printed and read back in via the usual Lisp print and read programs.

The Lisp routine **R** is the heart of **R-LOOP**. **R** takes a single s-expression as its input, translates it, reduces it with respect to **R-ALIST**, and returns either the special value **'(NOT REDUCIBLE)** or the result of introducing abbreviations back into the reduced value. The Lisp routine **S** takes two arguments. The first must be a Lisp symbol representing a variable in the logic and the second must be an s-expression whose translation is an explicit value. **S** modifies **R-ALIST** so that the value of the variable is the given explicit value.

12.53. REDUCE-TERM-CLOCK

General Form:

REDUCE-TERM-CLOCK

This Lisp variable may be set to any integer value. Its initial value is 100. The variable is used to prevent “infinite loops” in the executable counterparts of partial functions. In particular, the variable specifies the number of times such a function is permitted to recur before the execution is aborted.

For example, suppose that we introduce the function **APP** with

```
(DEFN APP (X Y)
  (EVAL$ T '(IF (EQUAL X (QUOTE NIL))
                Y
                (CONS (CAR X) (APP (CDR X) Y))))
  (LIST (CONS 'X X) (CONS 'Y Y)))
```

Then suppose we enter **R-LOOP** and try to evaluate **(APP '(A B C) '(1 2 3))**. The answer, **'(A B C 1 2 3)**, is quickly computed. However, if we try to evaluate **(APP 0 1)** we get the message: **APP aborted** and **R-LOOP** reports that the term is **(NOT REDUCIBLE)**.

Because **APP** was introduced without a proof of termination, its executable counterpart, ***1*APP**, is coded so that no more than **REDUCE-TERM-CLOCK** recursions are permitted. Because **(APP 0 1)** is undefined by the above recursion, the limit on the number of recursions is exceeded. Whenever the limit is exceeded the message **APP aborted** is printed by ***1*APP** and the term involving that computation is considered irreducible. While this seems perfectly sensible in the case of an infinite loop, like **(APP 0 1)**, it is less attractive in cases where **APP** is defined but the resource limitation imposed by **REDUCE-TERM-CLOCK** is insufficient to permit the computation to conclude. For example, **(APP '(1 2 3 ... 100) NIL)** cannot be computed if the **REDUCE-TERM-CLOCK** is set to 100.

Because the executable counterparts of defined functions are also called during proof attempts, the message that a computation has been **aborted** may appear in the middle of a proof attempt—even a successful proof attempt. This message is extremely annoying because it is not coordinated with the rest of the theorem prover's output and hence destroys the organization of that output. Furthermore, when the limit is set very high and a term involving an infinite loop occurs in the theorem, much time is spent in the needless attempt to evaluate the executable counterpart. In this case, time can be saved by setting **REDUCE-TERM-CLOCK** to some small positive integer—ideally to the depth of the maximum computation required in the proof. But we eschew such hackery because we don't like finding proofs only because of resource limitations. When we are tempted to play with the setting of **REDUCE-TERM-CLOCK**, we more often disable the executable counterpart of the function involved, after proving **REWRITE** rules that exhibit the values of the necessary computations.

If **REDUCE-TERM-CLOCK** is set to -1 then no attempt is made to limit the recursion. This may result in Lisp stack overflows during the proofs of theorems.

The value of **REDUCE-TERM-CLOCK** is not saved in library files created by **MAKE-LIB**. Thus, its value is not properly restored by **NOTE-LIB**. Neither is its value restored by **UNDO-BACK-THROUGH**. These omissions are essentially

due to the fact that we do not provide an event for manipulating the value of **REDUCE-TERM-CLOCK**, primarily because we think it would be so rarely used. In any case, the user is entirely responsible for maintaining the setting of **REDUCE-TERM-CLOCK** and failure to maintain it properly can conceivably prevent previously successful events from replaying.

12.54. REWRITE

REWRITE is one of the rule class tokens that may be given to **ADD-AXIOM**, **CONSTRAIN**, **FUNCTIONALLY-INSTANTIATE** or **PROVE-LEMMA** to indicate what kinds of rules should be generated from a formula. **REWRITE** rules actually come in four different forms: type prescription rules, compound recognizer rules, replacement rules, and linear arithmetic rules. All four forms are used in the simplification process, which is the first process tried. In addition, type prescription rules are used throughout the system to help determine the type of a term.

It is perhaps a mistake for us to lump together into the class “**REWRITE**” all four kinds of rules. Only replacement rules cause terms to be rewritten in the way traditionally associated with term rewrite systems. If a theorem built in as a **REWRITE** rule fails to cause rewriting, it may be that the rules generated from the theorem were not replacement rules.

We describe the simplification process starting on page 266. Starting on page 277 we describe how the four types of **REWRITE** rules are generated and how they are used. Starting on page 395 we give some hints regarding the use of **REWRITE** rules.

12.55. REWRITE-APPLY\$

REWRITE-APPLY\$ is the name of a special-purpose simplifier built into Nqthm. It simplifies some expressions of the form **(APPLY\$ (QUOTE fn) args)**. Roughly speaking, if **fn** is a symbol that names a total function (see page 291), then **(APPLY\$ (QUOTE fn) args)** is rewritten to **(fn (CAR args) ... (CAD...DR args))**, where the appropriate number of arguments for **fn** are taken from **args**. This simplification is justified by theorems proved in [30]. If the name **REWRITE-APPLY\$** is disabled, this simplifier is not used. The name is a citizen (a satellite of **GROUND-ZERO**) so it can be enabled and disabled. It is initially enabled.

12.56. REWRITE-APPLY-SUBR

REWRITE-APPLY-SUBR is the name of a special-purpose simplifier built into Nqthm. It simplifies some expressions of the form **(APPLY-SUBR (QUOTE fn) args)**. If **fn** is a function symbol, **(SUBRP 'fn)** is **T** and **fn** is not “erratic” as defined below, then **(APPLY-SUBR (QUOTE fn) args)** is rewritten to **(fn (CAR args) ... (CAD...DR args))**, where the appropriate number of arguments for **fn** are taken from **args**. If **(SUBRP 'fn)** is **F**, **(APPLY-SUBR (QUOTE fn) args)** is rewritten to **F**. These simplifications are justified by **SUBRP** axiom and **Axiom 55**.

The notion of “erratic” is quite subtle. At first sight, one is tempted to agree that **(APPLY-SUBR 'fn args)** is **(fn (CAR args) ... (CAD...DR args))**, if **(SUBRP 'fn)** is **T**. Certainly it is true for the **SUBRPs** that are introduced implicitly by our axiomatic acts, because we add the corresponding axiom about **APPLY-SUBR** as part of the “**SUBRP** axiom” (page 174). But it is consistent for the user to declare a new function symbol, say, **G**, of no arguments, and to add axioms specifying that **(SUBRP 'G)** is **T** but **(APPLY-SUBR 'G ARGS)** is **(NOT (G))**. Then clearly it would be incorrect to simplify **(APPLY-SUBR 'G args)** to **(G)**.

To avoid this we introduce the notion of “erratic” functions. A function symbol is *erratic* if it was introduced with **DCL** or **CONSTRAIN** or else mentions an erratic function in its **BODY**. Note that we sweepingly consider all declared functions erratic, rather than more carefully analyze the available **APPLY-SUBR** axioms.

If the name **REWRITE-APPLY-SUBR** is disabled, this simplifier is not used. The name is a citizen (a satellite of **GROUND-ZERO**) so it can be enabled and disabled. It is initially enabled.

12.57. REWRITE-CAR-V&C\$

REWRITE-CAR-V&C\$ is the name of a special-purpose simplifier built into Nqthm. It simplifies some expressions of the form **(CAR (V&C\$ T (QUOTE term) a))**. Roughly speaking, if **term** is a term containing no erratic symbols and the domain of the alist **a** is manifest, we simplify **(CAR (V&C\$ T (QUOTE term) a))** to the appropriate instance of **term**, provided **(V&C\$ T (QUOTE term) a)** is non-**F**, and to **0** otherwise. This simplification is justified by theorems proved in [30]. If the name **REWRITE-CAR-V&C\$** is disabled, this simplifier is not used. The name is a citizen (a satellite of **GROUND-ZERO**) so it can be enabled and disabled. It is initially enabled.

12.58. REWRITE-CAR-V&C-APPLY\$

REWRITE-CAR-V&C-APPLY\$ is the name of a special-purpose simplifier built into Nqthm. It simplifies some expressions of the form **(CAR (V&C-APPLY\$ (QUOTE fn) args))**. Roughly speaking, if **F** is not a member of **args** and **fn** is a total function other than **IF**, then **(CAR (V&C-APPLY\$ (QUOTE fn) args))** simplifies to **(fn (CAAR args) ... (CAAD...DR args))**. This simplification is justified by theorems proved in [30]. If the name **REWRITE-CAR-V&C-APPLY\$** is disabled, this simplifier is not used. The name is a citizen (a satellite of **GROUND-ZERO**) so it can be enabled and disabled. It is initially enabled.

12.59. REWRITE-EVAL\$

REWRITE-EVAL\$ is the name of a special-purpose simplifier built into Nqthm. It simplifies some expressions of the form **(EVAL\$ T (QUOTE term) a)** and some expressions of the form **(EVAL\$ T (CONS (QUOTE fn) args) a)**. For example, if **term** is tame and the domain of the alist **a** is manifest, then **(EVAL\$ T (QUOTE term) a)** is just a certain instance of **term**. This simplification is justified by theorems proved in [30]. If the name **REWRITE-EVAL\$** is disabled, this simplifier is not used. The name is a citizen (a satellite of **GROUND-ZERO**) so it can be enabled and disabled. It is initially enabled.

12.60. REWRITE-V&C\$

REWRITE-V&C\$ is the name of a special-purpose simplifier built into Nqthm. It simplifies some expressions of the form **(V&C\$ flg form a)**, where **flg** is **'LIST** or **form** is a **QUOTED** tame term. The simplification just replaces the expression by non-**F** in situations where propositional equivalence is sufficient. This is justified by theorems proved in [30]. If the name **REWRITE-V&C\$** is disabled, this simplifier is not used. The name is a citizen (a satellite of **GROUND-ZERO**) so it can be enabled and disabled. It is initially enabled.

12.61. REWRITE-V&C-APPLY\$

REWRITE-V&C-APPLY\$ is the name of a special-purpose simplifier built into Nqthm. It simplifies some expressions of the form **(V&C-APPLY\$ (QUOTE fn) args)**. For example, if **fn** is a total function other than **IF** and only propositional equivalence is of interest, then **(V&C-APPLY\$ (QUOTE fn) args)** is non-**F** provided **F** is not in **args**. This is justified by theorems proved in [30]. If the name **REWRITE-V&C-APPLY\$** is disabled, this simplifier is not used. The name is a citizen (a satellite of **GROUND-ZERO**) so it can be enabled and disabled. It is initially enabled.

12.62. Root Names

See the discussion of File Names on page 339.

12.63. Rule Classes

A *rule class* is one of the symbols **REWRITE**, **ELIM**, or **GENERALIZE** or else a list of the form **(META fn₁ ... fn_n)** where each **fn_i** is the name of a function in the logic. Rule classes are used to specify how a given axiom or theorem is to be stored as a rule in the data base. In Chapter 11 we describe in detail how rules are generated from formulas.

12.64. SET-STATUS

General Form:

```
(SET-STATUS name citizens action-alist)
```

Example Form:

```
(SET-STATUS CLOSE-OFF-LAYER1 T
  ((BOOT-STRAP INITIAL)
   (ADD-SHELL ENABLE)
   ((DEFN *1*DEFN) ENABLE)
   (OTHERWISE DISABLE)))
```


SET-STATUS is an event command that sweeps over a list of citizens and performs an action that may alter the enabled/disabled status of the citizen as a function of the type of event that introduced the citizen. See **DISABLE** for a general discussion of the effect and motivation behind enabling or disabling a name. See **DEFTHEORY** for a general discussion of the use of theories.

SET-STATUS does not evaluate its arguments. The first argument, **name**, must be a new name. The second argument, **citizens**, denotes some list of citizens and must be of one of the forms below:

- a theory name (as defined with **DEFTHEORY** or one of the built-in theory names **GROUND-ZERO** and **T**), denoting the value of the theory;
- a list of citizens denoting itself; or
- a pair of the form $(\mathbf{ev}_1 \text{ . } \mathbf{ev}_2)$ containing two event names, \mathbf{ev}_1 and \mathbf{ev}_2 , denoting all of the citizens introduced in the inclusive chronological interval bounded on one end by the event \mathbf{ev}_1 and on the other by \mathbf{ev}_2 .

Note that when an event pair is used, choice of order is unimportant. Finally, the third argument, **action-alist**, must be a list of doublets, each of the form $(\mathbf{key} \text{ } \mathbf{val})$, where each **key** is either a member of the set of words **{BOOT-STRAP ADD-SHELL DEFN *1*DEFN ADD-AXIOM PROVE-LEMMA CONSTRAIN FUNCTIONALLY-INSTANTIATE OTHERWISE}** or else is a proper list of members of that set, and each **val** is a member of the set **{ENABLE DISABLE AS-IS INITIAL}**, and where the following restrictions apply:

- no **key** word may occur twice in **action-alist**,
- the **val INITIAL** may be paired only with the **key BOOT-STRAP**, and
- the last element of **action-alist** must have **OTHERWISE** as its **key**.

The **action-alist** is used to determine a “status action” for citizens of the data base, as follows. If the citizen is the name of the executable counterpart of a function introduced with **DEFN**, then the status action is the **val** associated with the **key *1*DEFN** in **action-alist**, if there is one, and otherwise is the **val** associated with the **key OTHERWISE**. If the citizen is not such an executable counterpart, then its status action is the **val** associated with the kind of event command that created the citizen, if that **key** appears in the **action-alist**, and otherwise is the **val** associated with **OTHERWISE**.

SET-STATUS creates a new event, named **name**. The only other effect of **SET-STATUS** is possibly to change the enable/disable status of the citizens in the list denoted by **citizens**. The new status is determined by the status action for the name. If the action is **ENABLE**, the citizen is made enabled by **SET-STATUS**. If the action is **DISABLE**, the citizen is made disabled by **SET-STATUS**. If the action is **AS-IS** the citizen's status is unchanged. Finally, if the action is **INITIAL** then the citizen was created by **BOOT-STRAP** and its status is restored to what it was at the conclusion of **BOOT-STRAP**.

For example, the example command

```
(SET-STATUS EXAMPLE (PRIME-FACTORS . GROUND-ZERO)
  ((BOOT-STRAP INITIAL)
   (ADD-SHELL ENABLE)
   ((DEFN *1*DEFN) ENABLE)
   ((PROVE-LEMMA ADD-AXIOM) DISABLE)
   (OTHERWISE AS-IS)))
```

will create a new event named **EXAMPLE** and will possibly affect the status of all the names introduced in the inclusive interval from **GROUND-ZERO** through **PRIME-FACTORS**. In particular, it will leave all the satellites of **GROUND-ZERO** with the same status they had after the **BOOT-STRAP** that created them, it will enable all the satellites of all **ADD-SHELLs**, it will enable all defined functions and their executable counterparts, it will disable all **PROVE-LEMMAs** and **ADD-AXIOMs**, and it will leave all other citizens unchanged. Thus, for example, no **FUNCTIONALLY-INSTANTIATE** event is affected by this **SET-STATUS**

A common use of **SET-STATUS** is in its guise as **DISABLE-THEORY** and **ENABLE-THEORY**, where it is used to set the status of all the names in a given theory. We find that big proof projects often proceed in stages. During one particular stage, a certain collection of theories is heavily used, while in a later stage, a different collection of theories is needed. While one can manage this with local hints, it may be preferable (depending on the extent of each stage) to use global commands to configure the system to use the right rules automatically. At the end of a stage one then disables the now uninteresting theories and develops and enables the appropriate new ones.

In such projects we find useful such **SET-STATUS** commands as the following one.

```
(SET-STATUS END-OF-STAGE-1 T
  ((BOOT-STRAP INITIAL)
   ((ADD-SHELL DEFN *1*DEFN) ENABLE)
   (OTHERWISE DISABLE)))
```

This command essentially erects a barrier between one stage and the next. It restores the primitive rules (including shell rules) to their normal status, enables all definitions and executable counterparts, and disables everything else. Thus, the possibly hundreds of rules proved during “stage 1” are now inactive and we are ready to begin a completely new stage. If we find that we need rules that were proved during stage 1, we enable them explicitly when we discover they are needed. The reason we enable function symbols is that if we mention them in our subsequent work we will likely want them to behave as though we had just defined them; and if we do not mention them, their being enabled does not hurt us. Of course, we might want to **DISABLE** explicitly certain functions.

Warning. In an effort to be efficient, **SET-STATUS** does not touch the status recorded for a name unless it must change it. This has a bizarre interaction with **UNDO-NAME** that is best explained by illustration. First, forget about **SET-STATUS** and consider how **DISABLE** events (actually, **TOGGLE** events) affect the status of a name. Suppose **NAME** is initially enabled and that at event **a** it is disabled by (**DISABLE NAME**). Suppose some theorems are proved after **a** but no status events occur. Finally, suppose that another (**DISABLE NAME**) occurs at event **b**. Clearly, **NAME** is disabled after **b**. But suppose event **a** is undone with **UNDO-NAME**. What then is the status of **NAME**? The answer is that **NAME** is still disabled because event **b** disabled it and has not been undone or overridden.

Now consider the same scenario but this time use two **SET-STATUS** commands instead of the two **DISABLE** commands. For example, event **a** might be (**SET-STATUS a (NAME . NAME) ((OTHERWISE DISABLE))**) and **b** might be analogous. Because **NAME** is already disabled when event **b** is executed, event **b** has no effect on **NAME**. Thus, when **a** is undone with **UNDO-NAME** the status of **NAME** changes from disabled to enabled even though there is a subsequent **SET-STATUS** at event **b** that would have disabled it had event **b** actually been executed in the new context. We could have implemented **SET-STATUS** like **TOGGLE** so that it records the desired status regardless of the old status. We chose this implementation for efficiency: **SET-STATUS** is often used to map over thousands of names, many of which have already been disabled by previous sweeping **SET-STATUS** commands. We felt free to implement it this way, despite the difficulties with **UNDO-NAME**, because **UNDO-NAME** is already known not to maintain the integrity of the data base (see 321).

12.65. SET-TIME-LIMIT

General Forms:

(SET-TIME-LIMIT)

and

(SET-TIME-LIMIT n)

Example Form:

(SET-TIME-LIMIT 10)

SET-TIME-LIMIT is a Lisp routine that sets a limit on the amount of time that may be spent in the theorem prover. When the system notices that it has exceeded the limit, an error is signaled. When **n** is supplied, it must be a positive integer representing the time limit in seconds. **(SET-TIME-LIMIT n)** installs **n** as the limit. **(SET-TIME-LIMIT)** removes the limit. The system initially has no limit. The limit is not part of the data base and is not saved in library files. Thus, if you want a time limit during a given session, you must invoke **SET-TIME-LIMIT** in that session. Time limits are checked only during rewriting. There are other time-consuming loops in our system, such as “clausification.” Because of this, it is possible (but unusual) for the system to run considerably longer than the allotted time.

12.66. SKIM-FILE

General Forms:

(SKIM-FILE file)

and

(SKIM-FILE file :CHECK-SYNTAX cs-flg)

Example Form:

(SKIM-FILE "basic")

SKIM-FILE is a convenient way to call **PROVE-FILE** with the **:PETITIO-PRINCIPII** keyword argument set to **T** and the **:WRITE-FLAG-FILES** keyword argument set to **NIL**. **SKIM-FILE** prints a message to the terminal before calling **PROVE-FILE**; the message reminds the user that **SKIM-FILE** may render Nqthm inconsistent and that output is suppressed while skimming. See **PROVE-FILE** (page 356) for details and pay special attention to the warning about **:PETITIO-PRINCIPII**.

Roughly speaking, the effect of **SKIM-FILE** is to process all the event forms in **file**, merely *assuming* their admissibility. **SKIM-FILE** evaluates its arguments. The first argument, **file**, should be a file root name (see page 339) and **file.events**, the **events** extension of **file**, should exist and contain Nqthm event commands as described in the documentation of **PROVE-FILE**. The keyword argument is optional; its meaning is as described in **PROVE-FILE**.

12.67. THM Mode

THM mode refers to the set of axioms added by **BOOT-STRAP**. The system can be brought up so that it supports a constructive variant of the logic described here. This is in fact the logic supported by our system before we introduced the nonconstructive function **V&C\$** and the quantifier **FOR**. See the discussion of **BOOT-STRAP** on page 303.

12.68. Time Triple

A *time triple* is a triple of numbers printed at the end of event commands and indicating the amount of time used. The printed form of the triple is

[**pre-post prove output**]

where **pre-post** is the number of seconds spent verifying the syntactic preconditions of the command and generating and adding the appropriate rules to the data base, **prove** is the number of seconds spent in formal proof, and **output** is the number of seconds spent printing information to the user.

For example, in a **DEFN** command, **pre-post** includes the time taken to check that the body of the definition involves no free variables, etc., as well as the time to preprocess the definition to compute such information as the induction schemes suggested; **prove** is the time taken to prove the admissibility formulas; and **output** is the time taken to print the message associated with acceptance of a definition.

12.69. TOGGLE

General Form:

```
(TOGGLE new-name old-name flg)
```

Example Forms:

```
(TOGGLE TIMES-2-REWRITE-OFF TIMES-2-REWRITE T)
```

and

```
(TOGGLE REVERSE-ON REVERSE NIL)
```

TOGGLE is an event command for setting the status of a citizen in the data base. **new-name** must be a new event name, **old-name** must be a citizen, and **flg** should be **T** or **NIL**. The name of the new event created is **new-name**. The effect of the event is to set the status of **old-name** to **DISABLED** if **flg** is **T** and to **ENABLED** if **flg** is **NIL**. **TOGGLE** then prints a time triple and returns **new-name**.

The event commands **DISABLE** and **ENABLE** are both abbreviations for **TOGGLE** events. For details on the effect of disabling and enabling names, see the discussion of **DISABLE**.

12.70. TOGGLE-DEFINED-FUNCTIONS

General Form:

```
(TOGGLE-DEFINED-FUNCTIONS new-name flg)
```

Example Form:

```
(TOGGLE-DEFINED-FUNCTIONS DISABLE-ALL-COUNTERPARTS T)
```

TOGGLE-DEFINED-FUNCTIONS is an event command. Its arguments are not evaluated. Roughly speaking, this command allows you to disable, globally, all executable counterparts, i.e., prevent all functions from automatically evaluating on explicit values.

The first argument, **new-name**, must be a new event name. The second argument, **flg**, must be either **T** or **NIL**. The command sets the “executable counterparts disabled” flag of the data base to **flg**, creates a new event named **new-name**, prints a time triple, and returns **new-name**. The effect of the “executable counterparts disabled” flag is that, when non-**NIL**, no executable counterpart, other than shell constructors and base functions, are used during proofs. Thus, if the executable counterparts disabled flag is **T**, the expression **(PLUS 3 4)**, for example, would not simplify to **7**, except by repeated expansion of the definitional axiom or by **REWRITE** rules.

Technically speaking, the flag is not stored explicitly in the data base. Instead, the node for **new-name** is stored, and the most recently executed **TOGGLE-DEFINED-FUNCTIONS** event in the graph determines the value of the flag.

12.71. TRANSLATE

General Form:

(TRANSLATE x)

Example Form:

(TRANSLATE '(TIMES X Y (CADDR U)))

TRANSLATE is the Lisp procedure that maps user type-in into a well-formed term, or causes an error. The argument is evaluated. The value is then explored, abbreviations are expanded, and the result is checked for well-formedness. **TRANSLATE** returns the internal representation of the given term. We include **TRANSLATE** in this documentation simply so that you can experiment with the features of the Lisp reader to which you are typing.

In Appendix I and example file 10 we provide a formal definition of a function named **TRANSLATE**, corresponding to our notion in Chapter 4 of “well-formedness” and “translation.” The Lisp procedure **TRANSLATE** and the logical function **TRANSLATE** in Appendix I are closely related but are not identical. Both check for well-formedness. The Lisp procedure signals its absence with an error while the logical function returns **F**. But when the input is well formed, the Lisp translator returns a term in our internal form while the logical function returns the formal term it denotes. In our internal form, all explicit value terms are represented in **QUOTE** notation. But the whole purpose of the logical translator is to explicate that notation by exhibiting the (often huge) nest of formal constructors. Thus, when **'(ADD1 1)** is given to the Lisp translator, **(QUOTE 2)** is returned. But when **'(ADD1 1)** is given to the logical translator, **(ADD1 (ADD1 (ZERO)))** is returned.

12.72. UBT

General Forms:

(UBT)

and

(UBT x)

Example Forms:

(**UBT REVERSE**)

and

(**UBT 3**)

UBT is an abbreviated form of **UNDO-BACK-THROUGH**. The argument, **x**, is not evaluated and must be either an event name or a nonnegative integer. If **x** is an integer it defaults to the **x**th element of **CHRONOLOGY** (0-based counting, starting with the most recent event name). If no argument is provided, **x** defaults to the most recent event name. **UBT** removes from the data base every event from the most recent through **x**. The list of events forms removed, in chronological order, is stored in the global Lisp variable **UNDONE-EVENTS**. The list of events is suitable as the **events** argument to **PROVEALL** or **DO-EVENTS**; re-evaluating those events will reconstruct the data base. **UBT** returns the name of the oldest event removed, i.e., **x** or the name to which **x** defaulted.

UBT is guaranteed to restore the data base to the state it was in immediately before event **x** was created. Using **UBT** thus preserves the integrity of the data base. **UNDO-NAME** may not. See page 321.

12.73. UNBREAK-LEMMA

General Forms:

(**UNBREAK-LEMMA x**)

or

(**UNBREAK-LEMMA**)

Example Form:

(**UNBREAK-LEMMA 'REVERSE-REVERSE**)

The argument, **x**, is optional. If supplied, it is evaluated. If **x** is the name of a rule that is being monitored, it is removed from the list of monitored rules. See **BREAK-LEMMA**. If **x** is **NIL** or not supplied, all rules are removed from the list of monitored rules.

12.74. UNDO-BACK-THROUGH

General Form:

(UNDO-BACK-THROUGH *x*)

Example Form:

(UNDO-BACK-THROUGH 'REVERSE)

The argument, *x*, is evaluated and must be an event name. **UNDO-BACK-THROUGH** removes from the data base every event from the most recent through *x*. **UNDO-BACK-THROUGH** returns the list of event forms removed from the data base, in chronological order. The list returned is suitable as the **events** argument to **PROVEALL** or **DO-EVENTS**; re-evaluating those events will recreate the data base.

UNDO-BACK-THROUGH is guaranteed to restore the data base to the state it was in immediately before event *x* was created. Using **UNDO-BACK-THROUGH** thus preserves the integrity of the data base. **UNDO-NAME** may not. See page 321.

12.75. UNDO-NAME

General Form:

(UNDO-NAME *name*)

Example Form:

(UNDO-NAME 'REVERSE)

The argument, *x*, is evaluated and must be an event name. **UNDO-NAME** removes from the data base *x* and **ALL-DEPENDENTS** of *x* (see the discussion of **DATA-BASE** on page 319). **UNDO-NAME** returns the list of event forms removed from the data base, in chronological order. The list returned is suitable as the **events** argument of **DO-EVENTS**.

Wishfully speaking, **UNDO-NAME** removes from the data base *x* and all of its logical dependents. This is in fact often the case, but we do not guarantee that it is always the case. The problem, as noted on page 321, is that the theorem prover cannot track every use of certain kinds of rules and hence **ALL-DEPENDENTS** may not actually include all dependents. For this reason, **UNDO-NAME** prints a rather tiresome **WARNING** message every time it is called upon to remove any but the most recent event. **UNDO-NAME** is in fact correct when it removes only the most recent event and, more generally, when the events removed are those erased by **UNDO-BACK-THROUGH**.

Warning: Because **UNDO-NAME** may not maintain the integrity of the data base, we do not endorse any proof constructed in a session in which **UNDO-NAME** has been used. We discuss the notion of an “endorsed data base” on page 361.

Notwithstanding the foregoing caveat, it should be said that in day-to-day interaction with the theorem prover we make heavy use of **UNDO-NAME**. A common situation is to have defined a function, **fn**, differently than intended but not discover it until several dependent events (e.g., functions using **fn**) have been created. When we discover the discrepancy our usual response is to execute

```
(SETQ XXX (UNDO-NAME 'fn))
```

which undoes the introduction of **fn** and all of its dependents but leaves in place any events not dependent on **fn**. The Lisp variable **XXX** above is set to the list of events undone, starting with **fn**. Then we define **fn** “properly.” Finally, we execute

```
(DO-EVENTS (CDR XXX))
```

which attempts to reconstruct the dependents of the old **fn** on top of the new definition of **fn**. The **CDR** above is necessary to skip past the first event of **XXX**, which is the old definition of **fn**. Of course, the **DO-EVENTS** may not succeed: the new definition of **fn** may not have the right properties. More often when the **DO-EVENTS** fails it is because proofs constructed the first time those events were processed cannot be found now because of additions to the data base that were not undone. The most common situation is that names that were enabled when the original events were processed are now disabled, or vice versa. It sometimes happens that rules added to the data base since the original definition of **fn** now lead the theorem prover astray.

In general, we make free use of **UNDO-NAME**, behaving exactly as though it satisfied its wishful specification. However, after several days of such interactive work, and always upon reaching our final goal, we replay the entire list of events created since the last event in some endorsed data base. One way to “replay” an appropriate event sequence is to use **UNDO-BACK-THROUGH** to roll back to some endorsed event and simultaneously collect the event forms since then. Then **PROVEALL** or **DO-EVENTS** may be used to re-evaluate the events.

However, our standard way of ensuring that Nqthm endorses our final goal is to use **PROVE-FILE** on the script file that we build as the permanent record of the project. See page 257 for a description of how we use a text editor in conjunction with Nqthm to develop a permanent script and see **PROVE-FILE** (page 356) for a discussion of the issues surrounding “endorsement.” **PROVE-FILE** also creates a coherent file containing all the proofs.

We have never been burned by our use of **UNDO-NAME** and recommend its use provided its shortcomings are fully understood.

To illustrate how badly **UNDO-NAME** can burn you we offer the following command sequence which concludes with a proof that **T = F**.

```
(DEFN FN ( ) 23)

(PROVE-LEMMA FN-ARITY-IS-0 NIL (NLISTP (FORMALS 'FN)))

(UNDO-NAME 'FN)

(DEFN FN (X) 45)

(PROVE-LEMMA FN-ARITY-IS-NON-0 NIL
  (LISTP (FORMALS 'FN)))

(PROVE-LEMMA SURPRISE NIL (EQUAL T F)
  ((USE (FN-ARITY-IS-0)
    (FN-ARITY-IS-NON-0))))
```

The problem, of course, is that the **UNDO-NAME** above does not remove from the data base the theorem **FN-ARITY-IS-0** because its dependence on **FN** is not recorded since neither the statement of the theorem nor the proof involve the function symbol **FN**.

13 Hints on Using the Theorem Prover

This chapter is a loose collection of suggestions about how to use the theorem prover. Most of what is said here is not precise and there are exceptions to virtually every rule. This is unavoidable: if we could precisely describe successful techniques for using the system we would mechanize them.

The section headings of this chapter summarize the organization:

- How to Write “Good” Definitions
- How to Use REWRITE Rules
- How to Use ELIM Rules

13.1. How to Write “Good” Definitions

The theorem prover generally performs better on primitive recursive functions than on more complicated recursions. For example,

Definition.

```
(REVERSE X)
=
(IF (LISTP X)
  (APPEND (REVERSE (CDR X)) (LIST (CAR X)))
  NIL)
```

is generally more manageable than the smaller and computationally more efficient

Definition.

```
(REV1 X A)
=
(IF (LISTP X)
    (REV1 (CDR X) (CONS (CAR X) A))
    A).
```

Of course, the theorem prover can prove $(\text{REV1 } X \ A) = (\text{APPEND } (\text{REVERSE } X) \ A)$ and hence $(\text{REV1 } X \ \text{NIL}) = (\text{REVERSE } X)$, but it is prudent to arrange things so that **REVERSE** is the preferred form.

It is generally easier to state inductively proved properties of primitive recursive functions than properties of equivalent functions that accumulate intermediate results. The reason is that properties about functions of the latter type require consideration of arbitrary initial values of the accumulating arguments. For example, one must consider $(\text{REV1 } X \ A)$, not $(\text{REV1 } X \ \text{NIL})$. Users frequently forget to be sufficiently general. Furthermore, sufficiently general lemmas very rarely get produced by the system's own heuristics.

The system also has a performance bias towards definitions written as compositions of two or more recursive functions rather than those written as single complicated recursions. For example, in defining the optimizing expression compiler in [24], we optimized the expression in one pass and then generated compiled code for the optimized expression. To prove that the semantics of the expression were preserved by the composite compiler we had to prove two theorems, one for the optimizer and one for the code generator. We could have defined the compiler as a single function which did the optimization on the fly. Such functions may feel more efficient computationally, but intellectually they burden the proof process because one must address many different issues within the context of a single proof. If, for some reason, you want the top-level function to entangle several different issues, you should at least consider breaking the proof into pieces by defining the more elegant composite function and using it as a stepping stone to several isolated lemmas.

When a problem requires many different functions that recurse upon the same data structure, try to write each function so that it recurses in the same way. Put another way, adopt a given style and stick with it. This will pay off when the system does inductions on the data structure to unwind your functions.

Finally, the system is biased against large nonrecursive definitions. Non-recursive definitions are always expanded by the rewriter. Hence, no abstraction is gained by making such a definition, and conjectures using the concept are immediately transformed into large case analyses. The recommended procedure for dealing with such definitions follows. Define the concept. Prove the neces-

sary lemmas about it (usually as **REWRITE** rules). During this phase of the project, the definition buys you nothing because the concept is always replaced by its definition. When all the necessary lemmas are proved, **DISABLE** the concept. This prevents future use of the definition.

13.2. How to Use REWRITE Rules

Successful use of the theorem prover in even moderately interesting applications requires great care in the use of **REWRITE** rules. Experienced users of the system develop a “sense” of how to use **REWRITE** rules that is difficult to codify just yet. However, in this section we simply list a variety of hints one should keep in mind.

Almost all **REWRITE** rules are in fact replacement rules. Those that are not rarely seem to cause trouble. We therefore focus here entirely on the use of replacement rules. It is sometimes necessary to force a formula to generate a replacement rule even though it would otherwise generate a type prescription or linear arithmetic rule. To accomplish this, replace the conclusion, **concl**, by **(EQUAL concl T)**, thus removing the formula from the other two syntactic classes.

13.2.1. The Fundamental Fact: REWRITE Rules Rewrite

The most basic thing to keep in mind when you declare a formula to be a **REWRITE** rule is that it is (usually) a directive to replace some term by another when certain hypotheses rewrite to **T** (or, actually, to non-**F**). The directive generated is determined entirely by the syntax of the formula, in particular, the conditions you put in the hypothesis of the implication, the equality you put in the conclusion, and the orientation of that equality. (Without loss of generality here we assume all **REWRITE** lemmas conclude with an equality.)

Here is a simple mathematical truth: the last **CONS**-pair in the **REVERSE** of a nonempty list is the singleton list containing the first element of the list. Below are three equivalent ways of stating this fact:

1. **(IMPLIES (LISTP X)**
 (EQUAL (LAST (REVERSE X))
 (CONS (CAR X) NIL)))
2. **(IMPLIES (LISTP X)**
 (EQUAL (CONS (CAR X) NIL)
 (LAST (REVERSE X))))

3. $(\text{IMPLIES } (\text{NOT } (\text{EQUAL } (\text{LAST } (\text{REVERSE } X))$
 $(\text{CONS } (\text{CAR } X) \text{ NIL})))$
 $(\text{NOT } (\text{LISTP } X)))$

Note that the third formulation could be thought of as concluding with the equality $(\text{EQUAL } (\text{LISTP } X) \text{ F})$.

Consider the **REWRITE** rules generated by these three alternatives:

- **LAST-REVERSE-1**: replace every instance of $(\text{LAST } (\text{REVERSE } X))$ by $(\text{CONS } (\text{CAR } X) \text{ NIL})$ when $(\text{LISTP } X)$ rewrites to **T**.
- **LAST-REVERSE-2**: replace every instance of $(\text{CONS } (\text{CAR } X) \text{ NIL})$ by $(\text{LAST } (\text{REVERSE } X))$ when $(\text{LISTP } X)$ rewrites to **T**.
- **LAST-REVERSE-3**: replace every instance of $(\text{LISTP } X)$ by **F** when $(\text{EQUAL } (\text{LAST } (\text{REVERSE } X)) (\text{CONS } (\text{CAR } X) \text{ NIL}))$ rewrites to **F**.

Note how different the three induced behaviors are. When the relationship between **LAST** and **REVERSE** is formulated it is necessary that you contemplate how you want the information used.

Never declare a formula a **REWRITE** rule unless you have contemplated its interpretation as a directive.

13.2.2. Decide Upon a Rewrite Strategy

Replacement rules do two things: they can be used to put terms into normal form and they can be used to chain truths together. Selection of the hypothesis and conclusion literals determines what backward chaining is done. Orientation of the concluding equality determines the normal form.

It is generally advised to back chain from complex truths to simple ones. That is, one generally makes the hypotheses of a **REWRITE** rule simpler than the conclusion. For example, **LAST-REVERSE-3** above is an unusual rule because it means that every time the system encounters a **LISTP** expression it will back chain to a question about **LAST** and **REVERSE**. Since **LISTP** is more primitive, it is used in contexts in which **REVERSE**, say, is most likely an irrelevant concept. Theorems about sorting, about searching, about permutations, about 1:1 correspondences, about s-expressions representing terms or parse trees, etc., are unlikely to require reasoning about **REVERSE**.

As for orienting the concluding equalities, *it is generally advised to replace complex terms by simpler ones.* That is, the more complex expression should

generally be on the left-hand side of the concluding equality. For example, **LAST-REVERSE-2** is an unusual rule because it causes the system to replace every singleton list of the form **(CONS (CAR X) NIL)** by an expression involving **LAST** and **REVERSE**. In many contexts this would reformulate goal conjectures in terms of concepts that are only tenuously connected to the other concepts.

The above advice assumes an ordering on terms, under which some are “simpler” than others. Determining this ordering is exactly the problem faced by the **REWRITE** rule designer. However, whatever the order, it should be consistent with the ordering imposed by the subroutine hierarchy of function definitions in the data base. For example, since **LISTP** is used in the definition of **APPEND**, **LISTP** should be considered simpler than **APPEND**. Similarly, **APPEND** should be considered simpler than the definition of **REVERSE** which uses **APPEND**. The reason for this is that function definitions are used as replacement rules. Sometimes **REVERSE** expressions are replaced by expressions involving **APPEND**; sometimes **APPEND** expressions are replaced by expressions involving **LISTP**. If the user’s sense of “simplicity” runs counter to the definitional hierarchy, circular sets of rules are liable to result.

However, this advice does not give any hint for how to compare independent functions. For example

Definition.

```
(REV1 X A)
=
(IF (LISTP X)
    (REV1 (CDR X) (CONS (CAR X) A))
    A)
```

is an alternative definition of the “list reverse” function. In particular, **(REV1 X NIL)** is equal to **(REVERSE X)**. How do **APPEND** and **REVERSE** compare to **REV1** in simplicity? Several things can be said: **REV1** is simpler than **REVERSE** because it uses nothing but primitives while **REVERSE** uses the auxiliary concept **APPEND**. **REV1** is simpler than both **APPEND** and **REVERSE** because it is tail recursive and neither of the others are. **REV1** is more complicated than both because they are syntactically primitive recursive but **REV1** is not (it builds one of its arguments up as it recurses down the other).

Now, how should the equality

$$(\text{REV1 } X \text{ NIL}) = (\text{REVERSE } X)$$

be oriented? It boils down to the following question: would you prefer to see **REV1** or **REVERSE** in the intermediate goals generated by the system? The important fact to keep in mind is that you should adopt one of the two as the *preferred* expression of the concept of list reversal and use the preferred one in

all other rules. Thus, another rule in determining which of two functions is simplest is *consider \mathbf{f} simpler than \mathbf{g} if \mathbf{g} is a newly introduced function and there is a large body of **REWRITE** rules about \mathbf{f} in the data base.*

As the title of this subsection suggests, the real problem here is adopting a strategy for controlling the rewriter and applying it uniformly. Furthermore, the strategy must be consistent with the built-in tendency of the system to expand function calls by replacing them with their definitions. It is often difficult to decide upon a strategy early in the game. It is thus probably a mistake for the new user to spend a great deal of time making up general rules to follow. A more pragmatic approach is the one we use when embarking on new territory. First, as always, keep in mind how replacement rules are used. Second, as each new truth comes to light, decide how to orient it based on the previous decisions made and how it is needed in the proof at hand. Third, be prepared occasionally to stop and reorganize the rules entirely based on your newly gained familiarity with the concepts. Often the first two parts of this plan are sufficient to construct interesting proofs. The third part comes into play because occasionally in the exploration of a clique of concepts you will recognize the central role of a “new” or previously unappreciated concept. This most often happens because the new concept is strong enough to permit inductive proofs of theorems that then specialize to the cases you initially thought were important. When that happens you find yourself wanting your previously proved “main” theorems to be stated in terms of the new concept so everything is done with the same set of concepts.

Much of the following advice about **REWRITE** rules can be inferred from what has now been said. However, since we see many mistakes made repeatedly by virtually all new users, we will spell out additional advice.

13.2.3. Use Preferred Left-Hand Sides

Consider a rule like **LAST-REVERSE-1**, above, which rewrites instances of **(LAST (REVERSE X))**. Suppose that after **REV1** is introduced, it is adopted as the “preferred” concept, and **REVERSE** expressions are replaced henceforth by **REV1** expressions. Observe then that **LAST-REVERSE-1** would never again be used. Suppose the rewriter encountered the term **(LAST (REVERSE A))**, where **A** is a **LISTP**. At first sight, **LAST-REVERSE-1** applies. But recall that it is applied only after the interior terms have been rewritten. When **(REVERSE A)** is rewritten it will become **(REV1 A NIL)**. Thus, by the time **LAST-REVERSE-1** is tried, the term being considered will be **(LAST (REV1 A NIL))**, which does not match.

When formulating the left-hand side of a **REWRITE** rule you must make sure that all the proper subterms of the left-hand side are in preferred form. For

example, suppose you have decided to distribute **TIMES** over **PLUS**. Then it is useless to have a **REWRITE** rule whose left-hand side is **(REMAINDER (TIMES A (PLUS X Y)) A)** because no instance of that left-hand side will ever appear in a rewritten term. Put bluntly, if you have decided to distribute **TIMES** over **PLUS** you should write it that way in all your rules. That blunt advice is a little too strong: it is not necessary to use the preferred forms in the *right*-hand sides of replacement rules or in hypotheses that are free of free variables, because the rewriter doesn't look for those terms, it rewrites them. But rather than keep such caveats in mind it is easier simply to write everything in the preferred form.

Users seem frequently to forget that definitions are replacement rules and thus help determine the preferred form. For example, a left-hand side containing the term **(FIX X)**, as in the rule generated by

```
(EQUAL (NEXT-CLOCK (FIX X) Y)
        (FIX X))
```

is useless: any target term, **(NEXT-CLOCK (FIX a) b)**, matching the left-hand side of the rule, will have first been rewritten to

```
(NEXT-CLOCK (IF (NUMBERP a) a 0) b)
```

before the rule is tried. Thus, the rule will never apply because a match cannot be found.

In addition, the theorem prover contains many built-in rules about **IF**. Whenever **IF**s appear in a formula they cause case splits at the top level of the clause containing the **IF**.

Another common oversight has to do with the execution of functions on explicit values. Suppose **EXPT** has been defined to be the exponentiation function. Suppose the term **(EXPT 2 32)** arises in a proof, as in the term **(SUM A B (EXPT 2 32))**. This might happen if the term **(SUM A B (EXPT 2 WORDSIZE))** occurred in the user's conjecture but during the proof the case of **WORDSIZE** being **32** arises. Unless ***1*EXPT**, the executable counterpart of **EXPT**, is disabled, **(EXPT 2 32)** will rewrite to **4294967296**. If the user then finds the need to express a lemma about **(SUM A B (EXPT 2 32))** the uglier **(SUM A B 4294967296)** will have to be written, since **(EXPT 2 32)** will not ever appear in a rewritten term. This problem can be mitigated somewhat by the **IDENTITY** hack described on page 278. In particular, writing **(SUM A B (IDENTITY (EXPT 2 32)))** in a **REWRITE** lemma produces the same rule as would **(SUM A B 4294967296)**. In general, any time the left-hand side of a **REWRITE** lemma must contain an explicit value, the user has the option of writing **(IDENTITY term)**, where **term** is a variable-free term that reduces to the desired constant.

The general problem discussed here—of rewrite rules that apply to the left-hand sides of other rewrite rules—is a well-studied phenomenon in rewrite systems, called the *Knuth-Bendix* problem. The problem is first discussed in [88] where an algorithm for “completing” some sets of rewrite rules is presented. Since that seminal paper, many researchers have studied the ideas of term rewrite systems, extended senses of matching and unification, and improved or extended completion algorithms. For an excellent survey, see [69].

13.2.4. Avoid Circularities

The rewriter will “loop forever” on some sets of replacement rules when it is possible to rewrite some term **a** to **b** and **b** to **a**. This manifests itself as a stack overflow. Indeed, it is almost always the case that when a stack overflow occurs during a proof it is because of a circularity in the replacement rules.

Consider for example the rule generated from the formula

$$(\text{EQUAL } (\text{CARDINALITY } S) (\text{CARDINALITY } (\text{NORM } S))) .$$

The intuition behind this rule is that the user wants all occurrences of his or her **CARDINALITY** function to have “normalized” arguments. But the effect is that whenever the system encounters **(CARDINALITY x)** it rewrites it to **(CARDINALITY (NORM x))**, which it then rewrites to **(CARDINALITY (NORM (NORM x)))**, etc. Thus, even a single rule can cause infinite looping.

In a misguided attempt to “break” the loop, the user might prove as a **REWRITE** rule the formula

$$(\text{EQUAL } (\text{NORM } (\text{NORM } X)) (\text{NORM } X)) .$$

The idea is to “show the theorem prover that it is pointless to normalize more than once.” This attitude presumes a great deal on the theorem prover’s heuristics! With this rule in effect the loop becomes

$$\begin{aligned} & (\text{CARDINALITY } x) \\ &= (\text{CARDINALITY } (\text{NORM } x)) \\ &= (\text{CARDINALITY } (\text{NORM } (\text{NORM } x))) \\ &= (\text{CARDINALITY } (\text{NORM } x)) \\ &= (\text{CARDINALITY } (\text{NORM } (\text{NORM } x))) \\ &= \dots \end{aligned}$$

In general, loops can involve an arbitrary number of rules, including recursive definitions. Consider, for example, the rule generated from

$$(\text{EQUAL } (\text{ADD1 } (\text{PLUS } X Y)) (\text{PLUS } (\text{ADD1 } X) Y)) .$$

This rule “pushes **ADD1** into **PLUS**.” But the definition of **PLUS** is used in the

other direction. In particular, the system's use of the definition of **PLUS** (together with the rules for handling **IF**, **NUMBERP**, **EQUAL**, and **SUB1**) cause **(PLUS (ADD1 X) Y)** to split into two cases, according to **(NUMBERP X)**, and in one case to become **(ADD1 (PLUS X Y))**. If the rule for pushing **ADD1** into **PLUS** is available, the system then can loop.

It is difficult to characterize exactly when the system will loop because it has such an *ad hoc* collection of heuristics for controlling and sequencing the application of rules. But when a stack overflow occurs, it is most probable that a circularity has been hit. The only tool we have to help you track down circularities is **BREAK-REWRITE**, which allows you to inspect the rewrite path. Most often some hint of the circularity will appear there, but there are other recursive programs in our system (!) and it is possible the stack overflow—even one caused by replacement rules—will not show up on the rewrite path. See page 312.

13.2.5. Remember the Restrictions on Permutative Rules

Recall that the rule generated from

Theorem.

(EQUAL (PLUS I J)
(PLUS J I))

has the special restriction that, when applied, it does not produce a term that is lexicographically larger than the target term to which it was applied. Thus, **(PLUS B A)** will rewrite under the rule to **(PLUS A B)** but not *vice versa*.

The restriction applies more generally to any rule which permutes the variables in its left- and right-hand sides. The restriction is obviously intended to prevent circular rewriting while permitting some use to be made of commutativity. This can be used to normalize certain kinds of expressions, but care must be taken to arrange an appropriate set of rules.

It is a very common situation to have a function symbol that is both associative and commutative. Suppose, in addition to the commutativity of **PLUS** above we had the associativity, expressed as the rule generated from

Theorem.

(EQUAL (PLUS (PLUS I J) K)
(PLUS I (PLUS J K))).

Thus, any **PLUS** nest is right associated, and arguments are commuted within each **PLUS** to put them into lexicographically ascending order.

To what, then, does **(PLUS B (PLUS A C))** rewrite under these two rules? The obvious and desirable answer, namely **(PLUS A (PLUS B C))**,

is incorrect! Neither of the two rules can be applied, either at the top level or to the inner **PLUS** expression. The left-hand side of the associativity rule matches neither **PLUS** expression in the target. The left-hand side of the commutativity rule matches both, but neither application is permitted under the permutative restriction. In particular, either would move a lexicographically heavy term forward over a light term. That is, both $(\text{PLUS } (\text{PLUS } A \ C) \ B)$ (the result of commuting the outermost **PLUS**) and $(\text{PLUS } B \ (\text{PLUS } C \ A))$ (the result of commuting the innermost one) are lexicographically larger than the target. Thus, $(\text{PLUS } B \ (\text{PLUS } A \ C))$ is not rewritten by the two rules.

Of course, $(\text{PLUS } B \ (\text{PLUS } A \ C))$ is equal to $(\text{PLUS } A \ (\text{PLUS } B \ C))$ given associativity and commutativity; it is just that to derive the latter from the former it is necessary to “go against the grain” of the two rules and temporarily form $(\text{PLUS } (\text{PLUS } A \ C) \ B)$. Given our restrictions, though, how can such normalization be arranged?

The trick is to prove a third rule, derivable from the first two, namely

Theorem.

$(\text{EQUAL } (\text{PLUS } I \ (\text{PLUS } J \ K))$
 $\quad (\text{PLUS } J \ (\text{PLUS } I \ K)))$.

This is exactly what is needed above. More generally, any **PLUS** expression will be appropriately normalized now: the associativity rule will flatten the nest to the right, the new rule will sort all the addends but the last two into lexicographic order, and commutativity will take care of the last two.

13.2.6. *Eliminate Unnecessary Hypotheses*

It is often the case that new users encumber their **REWRITE** rules with logically unnecessary hypotheses. For example, while it is the case that

Theorem.

$(\text{IMPLIES } (\text{PROPERP } B) \ (\text{PROPERP } (\text{APPEND } A \ B)))$

a more powerful truth is

Theorem.

$(\text{EQUAL } (\text{PROPERP } (\text{APPEND } A \ B)) \ (\text{PROPERP } B))$.

The latter theorem has as advantages that it is easier to prove because less case analysis is done in the inductive proof, the generated rule is unconditionally applied, the rule eliminates the occurrence of **APPEND** altogether, and the rule introduces explicitly the question of whether **B** is proper whenever the question is raised about $(\text{APPEND } A \ B)$.

The latter advantage is particularly noteworthy. Suppose **(PROPERP (APPEND A (REVERSE B)))** is involved in a conjecture. Suppose you expect it to simplify to **T**, using the first rule above, but that the system fails to apply it. Then you must realize that the hypothesis of the rule failed to simplify to **T**. To make the rule apply you must figure out why the hypothesis failed and what rules should be added to make it simplify to **T**. All of this must be done without ever seeing the hypothesis in anything the system prints. In the case where the hypothesis undergoes elaborate simplification—but does not reduce to **T**—this is a very awkward problem (see **BREAK-LEMMA**). If, on the other hand, the system knows the second rule above, it is unconditionally applied, whether **(PROPERP (REVERSE B))** can be established or not. Regardless of what happens to **(PROPERP (REVERSE B))**, the result will be visible because it is part of the goal formula. It is thus much easier to see what rules are needed. Once the necessary rules have been found and **(PROPERP (REVERSE B))** rewrites to **T**, it hardly matters whether the relation between **PROPERP** and **APPEND** is phrased as a conditional or unconditional rule. But it is much easier to debug unsuccessful proof attempts when unconditional rewrite rules are used.

Sometimes the formulation of an unconditional replacement rule requires the introduction of an **IF** into the right-hand side of the rule. For example,

Theorem.

**(IMPLIES (NUMBERP N)
 (EQUAL (PLUS N 0) N))**

is less useful than

Theorem.

**(EQUAL (PLUS N 0)
 (IF (NUMBERP N) N 0))**

for all the reasons noted above.

To eliminate a hypothesis, consider what happens in the left-hand side when the hypothesis is violated and try to accommodate that in the right-hand side. This may require the introduction of an **IF** or some other function symbol into the right-hand side. Because our functions usually “behave” on “unexpected” inputs, it is frequently possible to remove mere type constraints on the arguments by substituting the coercion functions on the right-hand side (**FIX** could have been used above instead of the **IF**). We generally eliminate any hypothesis we can, provided it is not necessary to define a new recursive function to do it.

13.2.7. Use Free Variables Wisely

Recall that a free variable in a hypothesis is a variable that does not occur in the left-hand side nor in any previous hypothesis. To use such a rule, the rewriter must “guess” a suitable instantiation of the variable to relieve all the hypotheses containing it. Accidentally creating a rule with a free variable in it almost always causes trouble. Such rules are seldom applied successfully. However, our handling of free variables can be exploited by the informed user to make the rewriter more efficient.

One easy way to exploit free variables is to use them to prevent the rewriter from duplicating the work associated with simplifying common subexpressions. Recall that if the hypothesis containing the free variable, v , is of the form **(EQUAL v term)**, where **term** does not contain v , then the instantiation of v is chosen by rewriting **term** and binding v to the result.

Consider the rule generated from

```
(EQUAL (EXEC (APPEND A B) R M)
      (EXEC B
            (REG (EXEC A R M))
            (MEM (EXEC A R M))))).
```

Note that **(EXEC A R M)** occurs twice in the right-hand side. When the rewriter replaces **(EXEC (APPEND a b) r m)** by its counterpart, the expression **(EXEC a r m)** will be rewritten twice.

Consider now the rule generated from

```
(IMPLIES (EQUAL TEMP (EXEC A R M))
         (EQUAL (EXEC (APPEND A B) R M)
               (EXEC B (REG TEMP) (MEM TEMP))))).
```

This rule contains the free variable **TEMP**. Upon applying the rule to **(EXEC (APPEND a b) r m)**, the system rewrites **(EXEC a r m)**, binds the result to **TEMP**, and then rewrites **(EXEC B (REG TEMP) (MEM TEMP))** under the extended substitution. In this case, **TEMP** is rewritten twice, but each time it is simply a variable lookup.

If the hypothesis introducing the free variable is not of the form **(EQUAL v term)**, as above, then the situation is more complicated. Consider the rule generated by

```
(IMPLIES (MEMBER X (DELETE U V))
         (MEMBER X V))
```

The variable **U** is free in the hypothesis. Consider the attempted application of this rule to the term **(MEMBER e s)**, occurring in the test of an **IF**. The left-hand side of the rule is **(MEMBER X V)**, and the right-hand side is **T**.

Observe that when the left-hand side is matched against **(MEMBER e s)**, **x** is bound to **e** and **v** is bound to **s**. If the hypothesis can be established, the rule will replace **(MEMBER e s)** by **T**. Logically speaking, the formula permits such a replacement if there is a **U** such that **(MEMBER e (DELETE U s))**. Practically speaking, the rewriter must find some **U** with the hypothesized property.

It does this by searching through the known assumptions, looking for one of the form **(MEMBER e (DELETE d s))**. If such an assumption can be found, **U** is bound to **d** and the rule is applicable. If no such assumption is found, the rule is inapplicable. The important point to observe is that a hypothesis containing a free variable is not rewritten but is searched for among the assumptions. When a rule with free variables is generated, the system prints a warning message. Do not ignore these messages.

This state of affairs has its advantages and disadvantages. On the negative side, such hypotheses must be very carefully written if matches are to be found, since the assumptions, generally, are in rewritten form. A hypothesis containing both a free variable and an application of a nonrecursive function is likely never to be relieved, since the system will never find a call of a nonrecursive function in a rewritten assumption. This renders the entire rule useless. Given this restriction, it is perhaps surprising that rules containing free variables are ever used. Replacement rules with free variables in the hypotheses are often generated in response to a previously attempted proof that failed for want of that very rule. That is, in proving one's "main" result, a goal of the form **(IMPLIES (AND h₁ ... h_n) concl)** is adopted but not proved. Upon inspection at a check point the user notices that **(IMPLIES h_j concl)** is a theorem the system can prove. In this case it does not matter if the hypothesis contains free variables or not. The generated rule is applicable because the problematic hypothesis (in stable form) is literally among the hypotheses of the adopted goal.

On the positive side, the cavalier treatment of free variables severely limits the applicability of rules that otherwise would cause inefficiency. The previously mentioned rule

```
(IMPLIES (MEMBER X (DELETE U L))
  (MEMBER X L))
```

can be used to establish a completely arbitrary **MEMBERSHIP** relation. If great resources were used to establish the hypothesis, the rule would represent a terrible inefficiency since every single **MEMBER** expression triggers the attempt to relieve the hypotheses.

13.2.8. Bridge Lemmas—Hacking Around Knuth-Bendix

We have previously mentioned the “Knuth-Bendix” problem, which may be summarized as the situation in which a replacement rule fails to match a target term because some other rule has rewritten some interior subterm of the target.

Consider the function **TRUNC** that truncates a given bit-vector at a given position. It is a theorem that

Theorem. TRUNC-SIZE:
(IMPLIES (BITVP A)
(EQUAL (TRUNC A (SIZE A)) A)).

That is, truncating a bit-vector at its **SIZE** is a no-op. The rule generated from **TRUNC-SIZE** replaces any target of the form **(TRUNC a (SIZE a))** by **a** provided **(BITVP a)** can be established.

Now consider the target term

(TRUNC (APPEND B1 B2) (SIZE (APPEND B1 B2)))

to which we wish to apply **TRUNC-SIZE**. But suppose we have the rule

Theorem. SIZE-APPEND:
(EQUAL (SIZE (APPEND X Y))
(PLUS (SIZE X) (SIZE Y)))

which distributes **SIZE** over **APPEND**.

Since **SIZE-APPEND** is applied first (to the interior of the target term) the target becomes

(TRUNC (APPEND B1 B2)
(PLUS (SIZE B1) (SIZE B2)))

and **TRUNC-SIZE** no longer applies.

One way to avoid this problem is to **DISABLE SIZE-APPEND** or, in general, the rules that are causing the subterms of the target to rewrite. This may not be convenient if those same rules are used to advantage elsewhere in the main proof.

Another way is to provide a **USE** hint to the theorem prover when the main conjecture is submitted, instructing it to use the instance of **TRUNC-SIZE** obtained by replacing **A** by **(APPEND B1 B2)**. This is, perhaps, the preferred solution because it makes explicit in the record the fact that user intervention was required.

However, another way is to prove alternative forms of the lemmas in question, causing new replacement rules to be generated. These alternative forms have come to be called “bridge” lemmas because they bridge the gap between the terms that arise in applications and the more elegant terms used in the

statements of previously proved theorems. Unlike disabling and explicit **USE** hints, bridge lemmas have the advantage that they create general rules which can be used automatically in many proofs. But they look obscure and complicated, even though they are trivial consequences of other results. When we use bridge lemmas we generally tack the word **-BRIDGE** onto the name so that the genealogy of the lemma is understood.

Consider the **TRUNC-SIZE** rule above. A bridge lemma suitable for handling the case described is

Lemma. TRUNC-SIZE-APPEND-BRIDGE:
(IMPLIES (BITVP (APPEND X Y))
(EQUAL (TRUNC (APPEND X Y)
(PLUS (SIZE X)
(SIZE Y)))
(APPEND X Y))).

This is just the lemma obtained by instantiating **A** in **TRUNC-SIZE** with **(APPEND X Y)** and then applying **SIZE-APPEND**, i.e., it is one of the lemmas generated by applying the Knuth-Bendix superpositioning process to **TRUNC-SIZE** and **SIZE-APPEND**. To *prove* this theorem with the theorem prover it may be necessary to give the appropriate **USE** hint, but once proved the bridge lemma will be available without explicit hints.

A more powerful version of the bridge lemma is

Lemma. TRUNC-SIZE-BRIDGE:
(IMPLIES (AND (BITVP A) (EQUAL N (SIZE A)))
(EQUAL (TRUNC A N) A)).

This lemma is equivalent to **TRUNC-SIZE**, but it has the variable **N** where the **SIZE** expression was in the left-hand side and an equality hypothesis that **N** is that expression. But the rule generated from this lemma is much more powerful (more often applied) than that for **TRUNC-SIZE**. The left-hand side matches any **TRUNC** expression, causing the rewriter to backward chain to establish that the first argument is a bit-vector and that the second is provably equivalent to the **SIZE** of the first. If successful, the rewriter replaces the target with the first argument.

This rule will apply to the previously discussed rewritten target

(TRUNC (APPEND B1 B2)
(PLUS (SIZE B1) (SIZE B2))).

In particular, **A** is **(APPEND B1 B2)**, which we will call **a**, **N** is **(PLUS (SIZE B1) (SIZE B2))**, which we will call **n**, and when the rewriter tries to establish **(EQUAL n (SIZE a))**, **SIZE-APPEND** or *whatever rules* transformed **(SIZE a)** into **n**, will apply to establish the hypothesis.

While **TRUNC-SIZE** requires the second argument to be of the form **(SIZE a)** syntactically, the bridge version requires only that it be provably equal to **(SIZE a)**. It should be noted that the bridge lemma is inefficient: every **TRUNC** expression **(TRUNC a n)** establishes the temporary subgoal of showing that **(EQUAL n (SIZE a))**.

Bridges can be built, in general, by replacing some nonvariable proper subterm of the left-hand side by a new variable and adding the hypothesis that the new variable is equal to the subterm. This produces a more general, less efficient rule.

A problem with this rote construction of bridges is that it may introduce free variables. Consider

Theorem. PRIMES-DONT-DIVIDE-FACTS:
(IMPLIES (AND (PRIME P)
 (LESSP N P))
 (NOT (EQUAL (REMAINDER (FACT N) P)
 0)))

and its bridged version in which the interior **(FACT N)** is replaced by **X**:

Theorem. PRIMES-DONT-DIVIDE-FACTS-BRIDGE:
(IMPLIES (AND (PRIME P)
 (LESSP N P)
 (EQUAL X (FACT N)))
 (NOT (EQUAL (REMAINDER X P)
 0)))

Note that the bridge targets any term of the form **(EQUAL (REMAINDER x p) 0)** but has the free variable **N** which is chosen by instantiating **(LESSP N p)**. This bridge is thus not as generally applicable as it might appear.

13.2.9. Avoid Case Splits and Large Formulas

The theorem prover tends to do many proofs by cases. The reason is that every time an **IF**-expression is introduced into the conjecture, the conjecture is split into two parts according to whether the test of the **IF** is assumed true or false. The system is good at handling a large number of cases and so you should not give up a proof attempt simply because it has led to 20 or 30 cases. On the other hand, when a proof attempt has led to hundreds of cases, or cases are being spawned inside of cases inside of cases, etc., or the theorem prover simply is not printing anything (because the case explosion is so large that it is spending its time computing all the branches through the **IF**-tree), it is time to consider a new approach to your problem.

Most case explosions are caused by nonrecursive functions expanding. Therefore, it is generally advised to keep such functions disabled except when properties of their definitions are necessary. See the discussion of **DISABLE** in Chapter 12 and the discussion of preferred left-hand sides on page 398.

When the problem involves a hypothesis **(pred x)**, where the definition of **pred** is of the form

Definition.

(pred x) = (AND (p1 x) (p2 x) ... (pn x))

it is frequently convenient to disable **pred** after proving a sequence of lemmas of the form **(IMPLIES (pred x) (pi x))**. Thus, the **(pred x)** in the hypothesis of the main theorem is kept closed (thus saving the work associated with simplifying all of its conjuncts), but whenever one of its conjuncts is needed it is established by backchaining.

It is sometimes necessary to open a large function but desirable to wait until its arguments have been normalized. For example, **P** might be a function which takes a list as an argument and case splits on the **CAR** of the list, considering each of a finite set of possibilities. This frequently happens when **P** is an interpreter for some formal system. Suppose **P** is defined as

Definition.

(P X)
 =
(IF (EQUAL (CAR X) 'A) (A-STEP X)
(IF (EQUAL (CAR X) 'B) (B-STEP X)
(IF (EQUAL (CAR X) 'C) (C-STEP X)
...
(IF (EQUAL (CAR X) 'Z) (Z-STEP X)
(NO-STEP X))...)).

It frequently happens in these situations that in any particular theorem the **CAR** of **X** is equal to one of the known cases, but that it takes a bit of simplification to reduce it to the particular one. In such situations it is an excellent idea to disable **P** after proving a lemma of the form

(P (CONS (PACK U) V))
 =
(IF (EQUAL (PACK U) 'A) (A-STEP (CONS (PACK U) V))
(IF (EQUAL (PACK U) 'B) (B-STEP (CONS (PACK U) V))
(IF (EQUAL (PACK U) 'C) (C-STEP (CONS (PACK U) V))
...
(IF (EQUAL (PACK U) 'Z) (Z-STEP (CONS (PACK U) V))
(NO-STEP (CONS (PACK U) V))...)).

Observe that this is just the definition of **P** with the formal replaced by **(CONS (PACK U) V)**. The advantage of this rule is that **P** will stay closed until the argument expression is canonicalized to a list with some literal atom in its **CAR**. Only then will **P** expand with the above rule. If the canonical form chosen makes the **CAR** explicit, then exactly one case will come out of the nest of **IFs** above.

We make one final recommendation for dealing with large nests of **IFs**: rewrite rules can be fashioned for manipulating **IF**-trees *before* they reach the top of the formula and cause case splits.

For example, suppose that for each of the **-STEP** functions used in **P** we have proved that the predicate **INVP** holds. That is, suppose we have the sequence of lemmas

```
(INVP (A-STEP X))
(INVP (B-STEP X))
(INVP (C-STEP X))
...
(INVP (NO-STEP X))
```

and we wish to prove **(INVP (P X))**. The naive approach is to let **P** expand to a nest of **IFs**,

```
(INVP (IF (EQUAL (CAR X) 'A)
          (A-STEP X)
          (IF (EQUAL (CAR X) 'B)
              (B-STEP X)
              ...)))
```

then let the theorem prover explore the **IF**-tree and produce a set of 27 cases

```
(IMPLIES (EQUAL (CAR X) 'A)
          (INVP (A-STEP X)))
(IMPLIES (AND (NOT (EQUAL (CAR X) 'A))
              (EQUAL (CAR X) 'B))
          (INVP (B-STEP X)))
...
```

each of which will be proved by one of our lemmas.

A better approach is to prove the rewrite rule

```
(EQUAL (INVP (IF TEST X Y))
       (IF TEST
           (INVP X)
           (INVP Y))).
```

Using this rule the proof is much faster. During the initial rewrite, **P** is opened to the **IF**-nest, then the rule is applied 27 times in a linear pass to drive **INVP**

down to the tips of the **IF**-nest, and then each tip is reduced to **T** by one of our lemmas. Thus, the formula rewrites to **T** in a single pass.

Using similar rules for manipulating the **IF**-nests that arise in your conjectures, it is frequently possible to eliminate case splits altogether.

Here is a slightly more complicated example. Suppose we have another function, **R**, that is like **P** except that it uses **RA-STEP** for **A-STEP**, **RB-STEP** for **B-STEP**, etc. Suppose we have proved that $(P \rightarrow R (A\text{-STEP } X))$ is $(RA\text{-STEP } X)$, $(P \rightarrow R (B\text{-STEP } X))$ is $(RB\text{-STEP } X)$, etc. We wish to prove

$$(EQUAL (P \rightarrow R (P\ X)) (R\ X)).$$

The naive approach would be to let **P** and **R** each expand and let the system then handle the 27^2 paths generated. If the system does not exhaust its resources or the user's patience, it will eventually prove the theorem.

A better way is to prove the lemma that $P \rightarrow R$ can be driven through an **IF**, analogous to the lemma about **INVP** above. Then, after $(P\ X)$ expands into an **IF**-nest, the $P \rightarrow R$ walks through it and returns an **IF**-nest with $(P \rightarrow R (A\text{-STEP } X))$, etc., at the tips. Then let the $(R\ X)$ expand. It produces an isomorphic **IF**-nest in the second argument of the **EQUAL**.

It is at this point that we do something subtle. Instead of letting the system compute the 27^2 cases, consider the lemma

$$\begin{aligned} &(EQUAL (EQUAL (IF\ TEST\ P1\ P2) \\ &\quad (IF\ TEST\ R1\ R2)) \\ &\quad (IF\ TEST \\ &\quad\quad (EQUAL\ P1\ R1) \\ &\quad\quad (EQUAL\ P2\ R2))))). \end{aligned}$$

After the two arguments to the **EQUAL** have been rewritten to isomorphic **IF**-nests, the above rule drives the **EQUAL** down to the corresponding tips. It does not produce the cross-product of the cases because we anticipated that the same test arises from each **IF**-nest. Furthermore, once the **EQUAL** settles in at the tips, our previous lemmas relating $(P \rightarrow R (A\text{-STEP } X))$ and $(RA\text{-STEP } X)$, etc., apply and reduce each tip to **T**. Thus, in a single pass of the rewriter the entire theorem is reduced to **T**.

Note that this approach completely eliminated the case split in a situation where we might have otherwise had 27^2 cases. The key is the thoughtful definition of concepts (the parallel **IF** structure of **P** and **R**) and the considered use of rules to manipulate **IF**s.

13.3. How to Use ELIM Rules

Destructor elimination is important in domains where it is possible. Ask yourself “Can every object in this domain be uniquely represented as some combination of the outputs of the functions I’m using?” If so, it is likely you need an **ELIM** rule expressing that representability. If not, it is likely that you don’t need to think about **ELIM** at all.

Below is the most sophisticated **ELIM** theorem in common use in our data bases, the lemma that says that every natural **X** can be represented in terms of its remainder and quotient by a nonzero natural **Y**:

Theorem. REMAINDER-QUOTIENT-ELIM:
 (IMPLIES (AND (NUMBERP X)
 (NOT (ZEROP Y)))
 (EQUAL (PLUS (REMAINDER X Y)
 (TIMES Y (QUOTIENT X Y)))
 X))

Typical eliminations we arrange in our own data bases are as follows: eliminate **SUB1** in favor of **ADD1**, eliminate **DIFFERENCE** in favor of **PLUS**, eliminate **QUOTIENT** and **REMAINDER** in favor of **PLUS** and **TIMES**, eliminate **CAR** and **CDR** in favor of **CONS**, and eliminate **INITIAL-SEGMENT** and **FINAL-SEGMENT** in favor of **APPEND**. Destructor elimination is nice because by so representing the objects in your domain you can convert theorems about one set of function names into theorems about another and consequently focus on what otherwise seems like just half the problem.

Occasionally it is necessary to prove **GENERALIZE** rules in tandem with **ELIM** rules. For example, **REMAINDER-QUOTIENT-ELIM** causes **X** to be replaced by **R+Y*Q** in theorems involving **(REMAINDER X Y)** or **(QUOTIENT X Y)**. However, that replacement preserves the provability of the formula only if **R** is known to be less than **Y**. That constraint on **R** is provided by a **GENERALIZE** lemma such as

Theorem. REMAINDER-LESSP:
 (EQUAL (LESSP (REMAINDER X Y) Y)
 (NOT (ZEROP Y))).

Note that if **REMAINDER-QUOTIENT-ELIM** has been proved and stored as an **ELIM** rule before **REMAINDER-LESSP** is proved, an over-general elimination is performed during the proof attempt for **REMAINDER-LESSP**. The moral is simple:

When setting up the rules to arrange a destructor elimination that requires additional constraints on the new variables, prove the **GENERALIZE** theorems *before* you prove the **ELIM** theorem.

14 Installing Nqthm

In this chapter, we describe how to build an executable version of Nqthm.

14.1. The Source Files

For information about obtaining a copy of Nqthm, examine URL **ftp://ftp.cs.utexas.edu/pub/boyer/nqthm/README**, send an email message to **nqthm-users-request@cs.utexas.edu**, or write to the authors at the Computer Sciences Department of the University of Texas at Austin.

The first edition of *A Computational Logic Handbook* indicated that Nqthm was distributed without copyright or license. Although the current sources for Nqthm are now copyrighted and there is now a license, we still think of Nqthm as being freely available software. In fact, there are two licenses, and the user can freely choose under which license to operate.

Gnu License. All of the material in the Nqthm-1992 and subsequent distributions whose copyright is held by any of Boyer, Moore, or Computational Logic, Inc., is now licensed for public use under the terms of the Gnu General Public License (GPL). This licensing under the GPL covers all of the source files for Nqthm and many of the example files. We distribute with Nqthm, by permission, some example files that are in part or entirely the work of others, and this licensing under the GPL does not apply to those files. See the individual files under the **examples** subdirectory for specific licensing information for each such file. Most are in the public domain. The documentation subdirectory **doc** contains some information that is copyright by Academic Press and is dis-

tributed under the terms of a letter from Academic Press that may be found on that subdirectory. In particular, the **doc** subdirectory contains Chapter 4 and Chapter 12 of this book, providing on-line documentation for Nqthm users.

Nqthm License. Nqthm also may be used under its own special license, which is in some respects more liberal but in other respects less liberal than the GPL. That license may be found at the beginning of the source file **basis.lisp**, and in the files **nqthm-public-software-license.doc** and **nqthm-public-software-license.ps**.

Here is a list of the source file names. The files **defn-sk.lisp** and **infix.lisp** are not necessary to build Nqthm. Their utility is described later. The file **sloop.lisp** is a version of the MIT Loop facility written by Bill Schelter.

```

basis.lisp
code-1-a.lisp
code-b-d.lisp
code-e-m.lisp
code-n-r.lisp
code-s-z.lisp
defn-sk.lisp
events.lisp
genfact.lisp
infix.lisp
io.lisp
nqthm.lisp
ppr.lisp
sloop.lisp

```

Many other files are distributed with Nqthm, including example files (in the subdirectory **examples** and its subdirectories) and installation utilities (in the subdirectory **make**). However, only the above ***.lisp** files are necessary to build Nqthm from scratch.

The source files for Nqthm are written in the language Common Lisp, cf. *Common Lisp The Language* [131, 132]. Nqthm runs under implementations conforming to either [131] or [132]. We have run Nqthm under Gnu Common Lisp, Allegro Common Lisp, Lucid Common Lisp, CMU Common Lisp, Macintosh Common Lisp and Symbolics Common Lisp.

Sparc executable image shortcut. As a convenience to a large subset of potential Nqthm users, among the files that we distribute with Nqthm is an executable image of Nqthm built under Gnu Common Lisp for running on a Sparc. If you intend to run on a Sparc and are happy to use GCL, you can simply execute the file **sparc-gcl-nqthm-1992-image** and skip the

remaining installation instructions. Kyoto Common Lisp (KCL) was developed by Taiichi Yuasa and Masami Hagiya at Kyoto University. Gnu Common Lisp contains some enhancements to KCL and additional ports by Bill Schelter. Obviously, future changes to the Sparc architecture or operating system may render this image useless. In that case, please follow the instructions for building Nqthm from scratch below.

Unix make shortcut. Although compiling and loading Nqthm is relatively simple, compiling Lisp code so that it will execute with maximum speed and building an executable image are things that require a bit of Lisp expertise and whose details vary from Lisp to Lisp. If you will not be using the Sparc-GCL image that we distribute with Nqthm, but you will run under Unix and use one of GCL, Lucid, Allegro, or CMU Common Lisps, then we suggest that you next consider using the **make** command to build an executable Nqthm image. Simply connect to the directory containing the Nqthm sources and execute

```
% make LISP=xxx
```

where **xxx** is whatever command you use to invoke your Lisp. If successful, this command will compile Nqthm and create an executable file named **nqthm-1992** on the same directory. At successful conclusion, it will print out a message indicating that compilation, saving, and very simple testing of Nqthm-1992 worked ok. Subsequently and optionally, the tiny file **nqthm-1992** may be copied to a **bin** directory for easy invocation.

If this simple **make** command does not work, or if you are not running under Unix, you may easily build Nqthm with the very simple commands below.

To build Nqthm it is necessary to *compile* the source files and then to *load* the object files produced by the compilation.

14.2. Compilation

In order to compile Nqthm, you need to know how to start up your Common Lisp. You also need to have enough room on the directory with the source files to hold the compiled object files. Two megabytes of free disk space should suffice.

Loading the file `nqthm.lisp`. Under some operating systems, such as Unix, there exists the notion of the default directory associated with a process. When running under such an operating system, *connect* to the directory holding the Nqthm sources, for example by using the **cd** command, start up your Common Lisp, and execute the command (**LOAD "nqthm.lisp"**). On the other hand, if you are running under an operating system without such a notion, first start Common Lisp, then execute the Lisp command (**LOAD "...nqthm.lisp"**), where **...** is a suitable sequence of characters to iden-

tify the directory with the Nqthm sources, and then execute the command `(SETQ USER::*DEFAULT-NQTHM-PATH* "...")`.

Getting into the USER package. Next, execute the form:

```
(IN-PACKAGE "USER")
```

We emphasize that we really do mean package **"USER"**, not **"COMMON-LISP-USER"**, even on those Common Lisps that do not have a **"USER"** package initially. In Lisps that conform to the second edition of [132], there may initially be no **"USER"** package. However, loading of the file `nqthm.lisp` will then create a **"USER"** package. All Nqthm commands will be found in the **"USER"** package.

Setting the compiler optimization parameters. There is no single command we know that works across all Common Lisp versions and implementations to generate code that runs as fast as possible. Although it is not absolutely necessary, we recommend that you execute some command such as the following to generate efficient Nqthm compiled code.

```
(PROCLAIM
 (QUOTE (OPTIMIZE
         (SAFETY 0)
         (SPACE #+CMU 1 #-CMU 0)
         (SPEED 3)
         #+Allegro (DEBUG 0)
         #+(OR CMU Lucid) (COMPILATION-SPEED 0)
         #+CMU (EXTENSIONS:INHIBIT-WARNINGS 3))))
```

Compiling. Finally, execute

```
(COMPILE-NQTHM)
```

The compilation may take from a few minutes to an hour depending on the Lisp and machine you are using.

14.2.1. Example of Compilation

For example, to compile Nqthm under GCL on a Sun, assuming that the source files for the theorem prover have been placed on the directory `/usr/smith/nqthm-1992/`, we could execute these commands:

```
% cd /usr/smith/nqthm-1992
% gcl
> (LOAD "nqthm.lisp")
> (IN-PACKAGE "USER")
> (COMPILE-NQTHM)
```

Obviously, the first two commands are Unix commands (hence the % prompt) and the last three commands are fed to Lisp. We have omitted any compiler optimization parameter setting in this example because under GCL these parameters come set for maximum speed by default.

14.3. Loading

After Nqthm has been compiled, one may then *load* it into Common Lisp. To load Nqthm one merely (a) executes all of the commands above for compiling Nqthm except the final **(COMPILE-NQTHM)** command and then (b) executes **(LOAD-NQTHM)**. One may then begin using Nqthm commands, such as **(BOOT-STRAP NQTHM)**.

14.3.1. Example of Loading

Suppose that you have compiled Nqthm under Gnu Common Lisp on a Sun when connected to **/usr/smith/nqthm-1992/**. Then these commands will load Nqthm:

```
% cd /usr/smith/nqthm-1992
% gcl
> (LOAD "nqthm.lisp")
> (IN-PACKAGE "USER")
> (LOAD-NQTHM)
```

14.4. Executable Images

Once Nqthm-1992 has been compiled, building an executable image is generally a good idea because it makes starting up Nqthm very quick and convenient. Unfortunately, building an executable image requires a bit of expertise. See the file **make/save.lisp** for some examples of how to do this well under various Lisps.

14.5. Installation Problems

Although we have installed Nqthm on quite a variety of machines under quite a variety of Lisps, there are naturally problems that will arise.

One common problem is a shortage of disk space. This problem can arise during compilation, while saving an image, or while running examples. Unfortunately, one does not always get helpful diagnostic messages indicating that disk space is the underlying problem. For example, disk space shortage problems can appear as process death problems or as compilation problems. Here are some suggestions for how to begin to diagnose disk space problems. A good Lisp function to use to find out about Lisp space usage is **room**. Unix commands to use to help check for running out of disk space are **df**, **du**, and **pwd**. A Unix command to check virtual memory usage is **pstat**. A Unix command to check process size is **ps**. The latter two commands have to be executed while a process is running, of course, not after it has crashed.

If during the compilation (or loading) of Nqthm *collisions* between Nqthm's symbols and the symbols of the underlying Common Lisp are reported, then Nqthm may not work in the Lisp in question. For example, such a collision would occur if we were using GCL and had defined a function named **CLINES** in our Nqthm sources. **CLINES** is not listed in [131] or [132] and so we might believe it to be a name we could define. But in fact, in the current version of GCL, **CLINES** is defined as a function. Loading a definition of it provokes the message **Warning: CLINES is being redefined**. We suspect that we have finally eliminated all such collisions from the Lisps we use (Gnu, Allegro, Lucid, CMU, Macintosh, and Symbolics Common Lisps), but the variety of implementations and the ongoing development of Lisp leave us uncertain. In general, if any such collision is detected, you are advised not to trust the resulting Nqthm or Lisp. However, consultation with experts may well confirm that the collision is of no important consequence.

If the Lisp in question does not use the standard ASCII character codes for the printing ASCII characters (for example (**CHAR-CODE #\A**) is not 65), then Nqthm will not work. We check this during compilation.

Via Bill Schelter's **SLOOP** facility, we use fixnum arithmetic by default for certain iterations. Schelter's code is conservative and checks to make sure that numbers said to be fixnums are fixnums. Should a small range of fixnums result in an error report, one may work around this problem by setting **SLOOP::*AUTOMATIC-DECLARATIONS*** to **NIL** before compilation.

Proof-checking the entire set of Nqthm examples is a nontrivial data processing crunch. For example, if we run these tests on a Sparc Ultra under GCL, it takes about nine hours of cpu time. About 300 megabytes of files are created. Under some Lisps a process with a virtual memory size of over 87 megabytes may be created. Hence, your machine and your Lisp may be pressed to an

unusual extent with respect to the utilization of virtual memory, disk space, stack size, heap size, etc. Dealing with such problems is exceedingly operating system and Lisp implementation specific. To run example file 31 under GCL we need to enlarge the stack space allocated. We do this by (**SETF SI::*MULTIPLY-STACKS* 4**). GCL must return to its top-level read-eval-print loop after this command is executed in order for the command to take effect. Thus, this command cannot be included in the event file itself. On some Risc machines, e.g., Sparc, the GCL compiler may temporarily need over 100 megabytes of disk space when compiling some example ***.lisp** files. One solution to this is to cause GCL to split the source files into smaller chunks and to compile the chunks. See the documentation for the variable **COMPILER::*SPLIT-FILES*** in the GCL file **doc/DOC**. To run the Piton proofs, example file 31, under GCL, we execute (**SETQ COMPILER::*SPLIT-FILES* 200000**). Both the ***MULTIPLY-STACKS*** and the ***SPLIT-FILES*** settings are included both in the image file **sparc-gcl-nqthm-1992-image** and the file produced by the **make** command, under GCL.

Under some Lisps, there is likely to arise during compilation a very large number of warnings about functions that are called but not defined. These warnings may be ignored. The functions in question are defined in later files.

Version control can be a perilous problem. In the past, version control problems with Nqthm have been minor because, roughly speaking, each new version of Nqthm was issued in a different programming language: POP-2, Interlisp, Zetalisp, and Common Lisp. This chapter is concerned with the version of Nqthm known as Nqthm-1992, which is written in Common Lisp. The version of Nqthm that was distributed at the time of publication of the 1988 edition of the book *A Computational Logic Handbook* is now sometimes referred to as Nqthm-1987. Nqthm-1987 was also written in Common Lisp. To help distinguish between the 1987 and 1992 versions of Nqthm, we have arranged that each file of the Nqthm-1992 version of Nqthm contains the form (**CHK-BASE-AND-PACKAGE-1992 10 *PACKAGE***). No file of the 1987 version of Nqthm contains this form.

Help from the Nqthm-users group. An on-line mailing list for Nqthm users exists. News about Nqthm, interesting new applications, and advice about how to build and use the system are available via this mailing list. To join the mailing list, write to **nqthm-users-request@cs.utexas.edu**. To send a message to everyone on this list, write to **nqthm-users@cs.utexas.edu**. The archives for this mailing list are at URL **ftp://ftp.cs.utexas.edu/pub/boyer/nqthm/nqthm-users-mail-archive**.

14.6. Shakedown

As a very simple-minded test that the compilation and loading are successful, we recommend that you invoke (**BOOT-STRAP NQTHM**) and (**ASSOC-OF-APP**). The result should be a couple of pages of readable output, printed to the terminal, defining the function **APP** analogously to **APPEND** and proving its associativity. This only takes a few cpu seconds.

As a more thorough check that the theorem prover is running as we expect it to, we recommend that you use **PROVE-FILE-OUT** to process at least one of the many example files provided. A good choice is example file 13 (the file **examples/basic/proveall.events**), which corresponds closely to Appendix A of [24]. To run this particular test follow these two steps.

- Start Nqthm, either by running a save image or by loading Nqthm following the instructions above. Set the variable ***DEFAULT-NQTHM-PATH*** to a string (not a pathname) that names the **examples** directory for your copy of Nqthm. For example, on a Unix system you might execute

```
(SETQ *DEFAULT-NQTHM-PATH*
      "/usr/smith/nqthm-1992/examples/")
```

- Invoke the Lisp form (**PROVE-FILE-OUT "proveall"**). Evaluation of this form takes about one minute running under GCL on a Sparc Ultra, and should return **T**. The evaluation should also produce the files **proveall.proofs** (about 1.5 megabytes of proofs and definitions), **proveall.lib**, **proveall.lisp**, a compiled version of **proveall.lisp**, and the short text file **proveall.proved**, which you may read to check that evaluation was successful.

For a much more exhaustive test of Nqthm, a test representing many person-years of substantial Nqthm use, one may run the entire suite of examples provided on the subdirectories of the directory **examples**. To run this set of examples under a Unix for which the **make** instruction suggested above worked for Nqthm installation, see the instructions in the file **makefile**. On other systems, see the files **examples/driver.lisp** and **examples/README** for instructions.

14.7. Printing Nqthm Events in Conventional Notation

We provide a utility that will translate a file of Nqthm events into a LaTeX [91] or Scribe [51] source document. The utility, which may be found in **sinfix/sinfix.lisp**, also converts Nqthm formulas into conventional (“infix”) notation. For example, the Nqthm event

```
(prove-lemma boolean-sample (rewrite)
  (implies (and (lessp a b)
                (lessp b (length c)))
            (zerop (difference a (length c)))))
```

may be displayed as

THEOREM: boolean-sample
 $a < b \wedge b < \text{length}(c) \rightarrow a - \text{length}(c) \equiv 0$

when the LaTeX or Scribe produced by our utility is processed by the appropriate document preparation facility.

The main utility provided is **infix-file**, a Common Lisp routine which takes the name of an Nqthm event file (and some other arguments) and produces a corresponding LaTeX or Scribe file. Typically the user then includes this file in some larger LaTeX or Scribe document that contains the source text for a paper or presentation.

Because Nqthm event files frequently have Lisp-style comments in them (delimited by semicolons at the left and end-of-line at the right), **infix-file** copies comments to the output LaTeX or Scribe file, deleting the semicolons. Thus, the user can include text and formatting commands in the event file, but must know which document preparation facility—LaTeX or Scribe—will be used to process the text.

Unfortunately, “conventional” notation is remarkably unconventional as soon as one begins dealing with non-arithmetic operators (and even with arithmetic operators if one must distinguish, say, Peano addition from rational addition). For example, the user who is making heavy use of the notion of permutations may find it natural to display (**PERM X Y**) as “ $x \approx y$ ” while another user may be happier with the more conventional “perm(x, y).” Thus, our infix utility allows the user to specify how to display given operators. By appropriately loading the tables, sophisticated notation can be obtained from Nqthm events.

Here is a more elaborate example. The text below might be found in an Nqthm event file.


```

; @b<Example 2>: A more complex example showing indenting,
; processing of '!' formatting commands in comments, and
; extensions to the basic formatting functions. The
; functions !qTrains !qbig-delta and !qdelta2
; are defined to print special forms by the 'theory' file.
; The function !qTrains prints as a record selector.
; The constant function !qbig-delta prints as @g(D).
; The function !qdelta2 is given a relatively fancy print image.

(defn train-entry (j v a time ss)
  (implies (and (lessp (time (last ss)) time)
                (not (member j (trains (last ss))))
                (lessp (big-delta) (abs (delta2 j a v)))
                (block-not-in-use (car ts) ss))
    (state-seqp (append ss (update-state time j v a)))))

```

When this text is processed by **infix-file** and then the result is processed by Scribe the following is produced:

Example 2: A more complex example showing indenting, processing of '!' formatting commands in comments, and extensions to the basic formatting functions. The functions 'trains' 'big-delta' and 'delta2' are defined to print special forms by the 'theory' file. The function 'trains' prints as a record selector. The constant function 'big-delta' prints as Δ . The function 'delta2' is given a relatively fancy print image.

DEFINITION:

train-entry ($j, v, a, time, ss$)
 =

$$\begin{aligned} & \text{time}(\text{last}(ss)) < time \\ & \wedge j \notin \text{last}(ss).trains \\ & \wedge \Delta < |\Delta_j^a(v)| \\ & \wedge \text{block-not-in-use}(\text{car}(ts), ss) \\ \rightarrow & \text{state-seqp}(ss @ \text{update-state}(time, j, v, a)) \end{aligned}$$

Documentation for **infix-file** and its associated routines is provided in the file **sinfix/sinfix.lisp**.

The file **infix.lisp** contains an earlier implementation of **infix-file**, one that produces output for LaTeX, but not for Scribe. See the comments at the beginning of that file for instructions on its use. See the example files **examples/yu/mc20-1.events** and **examples/subramanian/mutilated-checkerboard.events**, together with the corresponding ***.ps** files, for examples of input to and output from this version of **infix-file**.

14.8. Example Event Files

We now describe the current contents of the **examples** directory. In each entry we give a file name, followed in parentheses by the author(s) and possibly a citation, followed by a very brief description of the work. When no citation is given, no published description of the work is available and the interested reader should look at the **events** file itself. Many of the files have explanatory comments. Each file has been successfully processed by **PROVE-FILE**. The files are listed in alphabetical order.

1. **examples/basic/alternating.events**

(Boyer) a formalization and correctness proof of the “Gilbreath Trick” [55, 58], a card trick having to do with the outcome of shuffling a deck of cards that has been previously arranged into alternating colors; the Nqthm attack on this problem was inspired by Gerard Huet’s use of the COQ theorem prover to do the proof [67, 68]

2. **examples/basic/async18.events**

(Moore, [105]) a model of asynchronous communication and a proof of the reliability of the biphase mark communications protocol

3. **examples/basic/binomial.events**

(Boyer and Moore, [30]) the binomial theorem expressed with **FOR** and a proof thereof

4. **examples/basic/controller.events**

(Boyer, Green, and Moore, [20]) a model of the problem of controlling a vehicle’s course and a proof that under certain conditions a particular program keeps the vehicle within a certain corridor of the desired course and, under more ideal conditions, homes to the course

5. **examples/basic/fibsums.events**

(Cowles) proofs of several interesting theorems about the sums of Fibonacci numbers

6. `examples/basic/fortran.events`

(Boyer and Moore, [26]) supporting definitions for a Fortran verification condition generator

7. `examples/basic/fs-examples.events`

(Boyer, Goldschlag, Kaufmann, and Moore, [19]) illustrations of the use of constrained functions and functional instantiation

8. `examples/basic/gauss.events`

(Russinoff, [121]) the original Nqthm proof of Gauss' law of quadratic reciprocity

9. `examples/basic/new-gauss.events`

(Russinoff, [121]) an improved proof of Gauss' law of quadratic reciprocity (after all, Gauss proved it eight times!)

10. `examples/basic/parser.events`

(Boyer and Moore, Appendix I of this book) a formalization of the syntax and abbreviation conventions of the Nqthm extended logic, expressed as a function from lists of ASCII character codes to the quotations of formal terms

11. `examples/basic/peter.events`

(Boyer) a sequence of lemmas describing the relationship between Ackermann's original function and R. Peter's version of it

12. `examples/basic/pr.events`

(Boyer) a proof of the existence of nonprimitive recursive functions

13. `examples/basic/proveall.events`

(Boyer and Moore, Appendix A of [24]) elementary list processing, number theory through Euclid's theorem and prime factorization, soundness and completeness of a tautology checker, correctness of the **CANCEL** metafunction, correctness of a simple assembly language program, correctness of a simple optimizing expression compiler

14. examples/basic/quant.events

(Boyer and Moore, [30]) illustrations of the use of **V&C\$** and **FOR**, including a study of several partial functions and functions, such as the “91 function,” that recurse on the value of their own recursive calls

15. examples/basic/rsa.events

(Boyer and Moore, [27]) proof of the invertibility of the Rivest/Shamir/Adleman public key encryption algorithm

16. examples/basic/small-machine.events

(Moore) an illustration of the use of simple operational semantics to prove program properties both directly and also via the so-called “functional” and “inductive assertion” methods

17. examples/basic/tic-tac-toe.events

(Moore) a formalization of what it means for a program to play nonlosing tic-tac-toe, the proof that a certain algorithm does so, and the successive refinement of the algorithm into the functional expression of an iterative number-crunching program

18. examples/basic/tmi.events

(Boyer and Moore, [28]) proof of the Turing completeness of Pure Lisp

19. examples/basic/unsolv.events

(Boyer and Moore, [29]) proof of the unsolvability of the halting problem for Pure Lisp

20. examples/basic/wilson.events

(Russinoff, [119]) proof of Wilson’s theorem

21. examples/basic/ztak.events

(Moore, [102]) proof of the termination of Takeuchi’s function

22. `examples/bevier/kit.events`

(Bevier, [3, 5, 6]) the formalization, implementation and proof that a simple separation kernel (implementing multiprocessing on a uniprocessor) provides process scheduling, error handling, message passing, and interfaces to asynchronous devices

23. `examples/bronstein/*.events`

(Bronstein and Talcott, [41, 40, 42, 43]) a collection of twenty six event files that are described in the four papers cited above; the work includes a formalization of “string-functional semantics” for circuit descriptions and the use of that semantics to verify the correctness properties of many circuits, including the Saxe-Leiserson retimed correlator, a pipelined ripple adder, and an abstract pipelined cpu

24. `examples/cowles/intro-eg.events`

(Cowles) a brief introduction to Nqthm intended for mathematicians and a proof of a theorem about factorial

25. `examples/cowles/shell.events`

(Cowles) alternative ways to decompose sequences and a study of Nqthm’s shell principle

26. `examples/flatau/app-c-d-e.events`

(Flatau, [53]) the development and proof of correctness of a compiler and runtime system for a subset of the Nqthm language (including **IF**, **CONS**, and subroutine call) requiring dynamic storage allocation; this event list corresponds to Appendices C, D, and E of [53] and deals with a runtime system that does not provide a garbage collector

27. `examples/flatau/app-f.events`

(Flatau, [53]) this event file is analogous to example file 26 but corresponds to Appendix F of [53] and deals with a runtime system including a reference counting garbage collector

28. **examples/fm9001-piton/big-add.events**

(Moore, [106]) a proof of the correctness of a Piton program for adding arbitrarily long numbers in base 2^{32}

29. **examples/fm9001-piton/fm9001-replay.events**

(Brock, Hunt, Kaufmann, and Young, [74, 38, 37]) statement and proof of the correctness of the gate-level implementation of the FM9001 microprocessor. See also the files in the directory **examples/fm9001-piton/fm9001** for more easily digestible descriptions of the parts of and the proofs about the FM9001 microprocessor. For an overview of the FM9001 project, see URL **ftp://ftp.cs.utexas.edu/pub/boyer/fm9001/index.html**

30. **examples/fm9001-piton/nim-piton.events**

(Wilding, [149, 150]) a proof that a given 300-line Piton program plays the game of Nim (see page 10) optimally; the program is also shown to be loadable onto the FM9001 (satisfying the requirements of the correctness theorem for Piton); bounds on the program's execution time have been proved using Pc-Nqthm [75, 21]. Also of very considerable interest is Wilding's further work [152, 151] on real-time programs with Nqthm as the prover

31. **examples/fm9001-piton/piton.events**

(Moore, [106]) the definition of the Piton assembly language, its implementation on the FM9001 via a compiler, assembler and linker, and a proof of the correctness of the FM9001 implementation

32. **examples/fortran-vcg/fortran.events**

(Boyer and Moore, [26]) the same file as **basic/fortran**, above, which is duplicated on this sub-directory for technical reasons

33. **examples/fortran-vcg/fsrch.events**

(Boyer and Moore, [26]) proofs of the verification conditions for a Fortran implementation of the Boyer-Moore fast string searching algorithm

34. **examples/fortran-vcg/isqrt.events**

(Boyer and Moore, [20]) proofs of the verification conditions for a Fortran implementation of the integer version of Newton's square root algorithm

35. **examples/fortran-vcg/mjrty.events**

(Boyer and Moore, [32]) proofs of the verification conditions for a Fortran implementation of a linear-time majority vote algorithm

36. **examples/hunt/fm8501.events**

(Hunt, [73, 70]) formalizations of the machine code for the 16-bit FM8501 microprocessor, a register transfer model of a microcoded implementation of the machine, and a proof of their correspondence

37. **examples/kaufmann/expr-compiler.events**

(Kaufmann, see Young [155]) the proof of correctness of a simple expression compiler, designed as an exercise for beginners

38. **examples/kaufmann/foldr.events**

(Kaufmann) an illustration of a method of proving permutation-independence of list processing functions

39. **examples/kaufmann/generalize-all.events**

(Kaufmann, [80]) the correctness of a generalization algorithm that operates in the presence of free variables

40. **examples/kaufmann/koenig.events**

(Kaufmann, [83]) a proof of Koenig's tree lemma

41. **examples/kaufmann/locking.events**

(Kaufmann, [81]) a model of a simple data base against which read and write transactions can occur

42. **examples/kaufmann/mergesort-demo.events**

(Kaufmann) the correctness of a merge sort function

43. **examples/kaufmann/note-100.events**

(Kaufmann, [79]) the proof of Ramsey's theorem for exponent 2, finite case, described in a style intended to assist those wishing to improve their effectiveness with Nqthm

44. **examples/kaufmann/partial.events**

(Kaufmann) an approach to handling partial functions with Nqthm

45. **examples/kaufmann/permutationp-subbagp.events**

(Kaufmann) a formalization of the notion of permutation via bags

46. **examples/kaufmann/ramsey.events**

(Kaufmann, [83]) a proof of Ramsey's theorem for the infinite case

47. **examples/kaufmann/rotate.events**

(Kaufmann, [77]) a proof about rotations of lists, intended as an introduction to Nqthm

48. **examples/kaufmann/rpn.events**

(Kaufmann and Jamsek) an exercise in reverse Polish notation evaluation

49. **examples/kaufmann/shuffle.events**

(Kaufmann) another solution to the Gilbreath card trick challenge (see example file 1)

50. **examples/kunen/ack.events**

(Kunen) an illustrative definition of Ackermann's function

51. **examples/kunen/induct.events**

(Kunen, [90]) shows that induction on ω^2 is provable in Primitive Recursive Arithmetic and gives a simple demonstration of the nonconstructiveness of **EVAL\$** and **V&C\$**

52. **examples/kunen/new-prime.events**

(Kunen) an alternative proof of the fundamental theorem of arithmetic that—unlike the one presented in [24]—does not use concepts not involved in the statement of the theorem

53. **examples/kunen/paris-harrington.events**

(Kunen, [89]) a proof of the Paris-Harrington Ramsey theorem, and some related results

54. **examples/numbers/arithmetric-geometric-mean.events**

(Kaufmann and Pecchiari, [85]) the arithmetic-geometric mean theorem

55. **examples/numbers/bags.events**

(Bevier) a library of useful definitions and lemmas about bags

56. **examples/numbers/extras.events**

(Wilding) a trivial extension of the **integers** library used in **fib2** below

57. **examples/numbers/fib2.events**

(Wilding, [148]) a proof of Matijasevich’s lemma about Fibonacci numbers

58. **examples/numbers/integers.events**

(Bevier, Kaufmann, and Wilding, [78]) a library of useful definitions and lemmas about the integers

59. **examples/numbers/naturals.events**

(Bevier, Kaufmann, and Wilding, [4]) a library of useful definitions and lemmas about the natural numbers

60. **examples/numbers/nim.events**

(Wilding) a formalization of the game of Nim and a proof that a certain algorithm implements a winning strategy

61. **examples/numbers/scheduler.events**

(Wilding, [151]) a proof of the optimality of an earliest-deadline-first scheduler on any set of periodic tasks

62. **examples/numbers/tossing.events**

(Kaufmann) a simple proof about coin tossing

63. **examples/shankar/church-rosser.events**

(Shankar, [128, 127]) a proof of the Church-Rosser theorem for lambda-calculus

64. **examples/shankar/goedel.events**

(Shankar, [127, 129]) a proof of Gödel's incompleteness theorem for Shoenfield's first order logic extended with Cohen's axioms for hereditarily finite set theory, Z2

65. **examples/shankar/tautology.events**

(Shankar, [126, 127]) a proof that every tautology has a proof in Shoenfield's propositional logic

66. **examples/subramanian/mutilated-checkerboard.events**

(Subramanian, [134, 133]) proof that a mutilated checkerboard cannot be tiled. See also the file **examples/subramanian/mutilated-checkerboard.ps**

67. **examples/talcott/mutex-atomic.events**

(Nagayama and Talcott, [108]) a proof of the local correctness of a mutual exclusion algorithm under a certain "atomicity assumption"

68. **examples/talcott/mutex-molecular.events**

(Nagayama and Talcott, [108]) a proof of the local correctness of a mutual exclusion algorithm without the "atomicity assumption" mentioned above

69. **examples/young/train.events**

(Young, [157]) models and proves a theorem about a railroad train, a gate, and a controller

70. **examples/yu/amax.events**

(Yu) the correctness proof for the MC68020 machine code produced by the Gnu C compiler for a C program that finds the maximum value in an integer array

71. **examples/yu/asm.events**

(Yu, [159, 35]) the correctness proof for the MC68020 machine code produced by the Gnu C compiler for a trivial C program that uses embedded assembly code (the object being to demonstrate that embedded assembly code can be handled)

72. **examples/yu/bsearch.events**

(Yu, [159, 35]) the correctness proof for the MC68020 machine code produced by the Gnu C compiler for a binary search program written in C

73. **examples/yu/cstring.events**

(Yu, [159, 35]) the correctness proofs for the MC68020 machine code produced by the Gnu C compiler for 21 of the 22 C String Library functions from the Berkeley Unix C string library; the proof for each function is broken into two “phases”; the first phase establishes the correspondence of the machine code and a suitable recursive function and the second phase establishes that the recursive function has the specified properties; the file **examples/yu/cstring.events** actually contains the second phase proofs for all of the string functions handled; the first phase proof for each string function is contained in a separate **events** file named for the string function, e.g., **examples/yu/memchr.events**, **examples/yu/memcmp.events**, **examples/yu/strnspn.events**, etc.

74. **examples/yu/fixnum-gcd.events**

(Yu) the correctness proof for the MC68020 machine code produced by the GCL compiler for a Common Lisp program that computes the greatest common divisor of two **FIXNUMS**

75. `examples/ym/fmax.events`

(Yu, [159, 35]) the correctness proof for the MC68020 machine code produced by the Gnu C compiler for a trivial C program to compute the maximum of two integers according to a supplied comparison function (the object being to demonstrate that C “function pointers” are handled)

76. `examples/ym/gcd.events`

(Yu, [159, 35]) the correctness proof for the MC68020 machine code produced by the Gnu C compiler for Euclid’s greatest common divisor algorithm written in C

77. `examples/ym/gcd3.events`

(Yu, [159, 35]) the correctness proof for the MC68020 machine code produced by the Gnu C compiler for a C program consisting of two nested calls of the GCD program (the object being to demonstrate that procedure call is handled in a way that allows hierarchical verification)

78. `examples/ym/group.events`

(Yu, [158]) proofs of two theorems in finite group theory, the first about kernels of homomorphisms and the second about the Lagrange theorem

79. `examples/ym/isqrt.events`

(Yu) the correctness proof for the MC68020 machine code produced by the Gnu C compiler for C program for computing integer square roots via Newton’s method

80. `examples/ym/isqrt-ada.events`

(Yu) the correctness proof for the MC68020 machine code produced by the Verdix Ada compiler from an Ada program for computing integer square roots via Newton’s method

81. `examples/ym/log2.events`

(Yu) the correctness proof for the MC68020 machine code produced by the Gnu C compiler for a C program for computing integer logarithms (base 2) via repeated division by 2

82. `examples/yu/mc20-0.events`

(Yu, [159, 35]) some utilities involved in the formal specification of the MC68020

83. `examples/yu/mc20-1.events`

(Yu, [34, 35]) the formal specification of about 80% of the user available instructions for the Motorola MC68020 microprocessor

84. `examples/yu/mc20-2.events`

(Yu, [159, 35]) a library of useful definitions and lemmas about the formalization of the MC68020

85. `examples/yu/mjrty.events`

(Yu) the correctness proof for the MC68020 machine code produced by the Gnu C compiler for a linear time majority vote algorithm written in C

86. `examples/yu/qsort.events`

(Yu) the correctness proof for the MC68020 machine code produced by the Gnu C compiler for Hoare's *in situ* quick sort program written in C; the source code is that on page 87 of [86] except that inline code is used rather than the subroutine **swap**; Yu reports (personal communication) that this change was made only because he was, at the time, investigating methods of proving recursive programs correct and did not want to be distracted by other subroutine calls

87. `examples/yu/switch.events`

(Yu, [159, 35]) the correctness proof for the MC68020 machine code produced by the Gnu C compiler for a trivial C program that employs the **switch** statement (the object being to demonstrate the technique used to compile **switch** can be handled)

88. `examples/yu/zero.events`

(Yu) the correctness proof for the MC68020 machine code produced by the Gnu C compiler for a C program that zeros an array, illustrating the handling of writes to memory

14.9. DEFN-SK

The file **defn-sk.lisp**, which is distributed with the Nqthm sources, implements a *Skolemizer*. Via this facility, one can handle first order logic quantifiers within Nqthm. See Kaufmann's paper [83] for further details. See the beginning of the file **defn-sk.lisp** for instructions. This facility is the work of Matt Kaufmann, and he has incorporated it into his Pc-Nqthm system. We include **defn-sk.lisp** with the distribution of Nqthm because it is used in the following example event files in subdirectory **examples/***yu*: **cstring**, **qsort**, and **group**.

Appendix I

A Parser for the Syntax

Below we reproduce verbatim example file 10 in which we formalize the extended syntax in the logic itself. We offer this event list as

- an alternative equivalent specification of the extended syntax,
- an example of formalization in the logic,
- an example of a file suitable for processing by **PROVE-FILE**, and
- an example of the tactics used to admit complicated definitions.

Furthermore, the data base created by processing this file allows the Nqthm user to experiment with the syntax, i.e., to obtain the formal term denoted by the display of any well-formed term in the extended syntax.

All of the text below is new to the second edition.

```
; The Formalized Extended Syntax
```

```
; SECTION: Introduction
```

```
; This file, /examples/basic/parser.events, is a formalization of  
; the extended syntax as presented in Chapter 4. We formalize the  
; syntax in the logic itself. Roughly speaking, for every occur-  
; rence of ``Terminology'' in Chapter 4 there is an admissible  
; function definition here that formalizes the concept defined in  
; Chapter 4. For example, we formally define what it is to be a
```



```
; ``numeric sequence'' by introducing a function which returns T or
; F according to whether its argument is such a sequence.
```

```
; The outline of our presentation is as follows. We first develop
; the notion of a ``token tree.'' Token trees in the logic are re-
; presented by integers, literal atoms, and list structures. The
; text in Chapter 4 then develops the idea of how to ``display'' a
; token tree. Here we do the inverse: we formalize the notion of
; how to parse a token tree from a sequence of ASCII character
; codes. This culminates in the definition of READ-TOKEN-TREE
; which takes a list of character codes and returns either F (in-
; dicating that the input sequence is unparsable) or a token tree.
; We then return to the text of Chapter 4 and formalize what it is
; for a token tree to be ``readable,'' what ``readmacro-expansion''
; is, and what an ``s-expression'' is. This part of the formal-
; ization may illuminate backquote notation for those unfamiliar
; with it. We finally define what it is for an s-expression to be
; ``well-formed'' and what the ``translation'' of such an
; s-expression is.
```

```
; This presentation is meant to clarify the extended syntax. Thus,
; it would hardly do to express the formalization in terms of the
; extended syntax. We therefore largely confine ourselves here to
; the formal syntax with the following exceptions:
```

```
; 1. We permit ourselves to write natural numbers, e.g., 2, in
; place of their formal equivalents, e.g., (ADD1 (ADD1 (ZERO))).
```

```
; 2. We permit ourselves to use QUOTE notation to denote literal
; atom constants. Thus, we write 'ABC where
; (PACK (CONS 65 (CONS 66 (CONS 67 0))))
; would otherwise be needed.
```

```
; 3. We occasionally display list constants in QUOTE notation,
; e.g., '(UPPER-B UPPER-O UPPER-X) is written in place of
; (CONS 'UPPER-B (CONS 'UPPER-O (CONS 'UPPER-X 'NIL))). Two
; large association lists are displayed with QUOTE notation.
; These displays are in conjunction with the definitions of
; utility functions whose semantics are intuitively clear,
; namely a function that maps from our name for an ASCII
; character to its ASCII code and a function that maps from the
; name of a function to its arity.
```

```
; 4. We permit ourselves to write comments.
```

```
; It should be stressed that these are ``exceptions'' only in the
; sense that they are not part of the formal syntax. These con-
; ventions are entirely supported by the extended syntax and we
; note their use only because they are exceptions to our self-
; imposed restriction to the formal syntax in this file.
```

```
; We have followed the text's style of the definition very closely.
; This results in a somewhat ``inefficient'' formalization. For
```

```
; example, it is possible to implement the parsing/readmacro-expan-
; sion process in a single pass, but we have not done so. While we
; feel our use of the logic here is illustrative of ``good usage''
; the critical reader must keep in mind that we are trying to
; formalize the ideas as presented in the text and not merely code
; a parser in the logic.
```

```
; This file may be processed with PROVE-FILE and the resulting
; library may be noted and used in R-LOOP to experiment with the
; syntax. We recommend that the library be compiled. If the Nqthm
; installer has so processed the file, the library file may already
; exist as /examples/basic/parser. This source file also contains
; ``commented out'' Common Lisp code that permits more convenient
; experimentation. Search for occurrences of ``defun'' to find the
; two regions in question.
```

```
; Finally, this file serves as a good example of a fairly substan-
; tial Nqthm formalization effort. We recommend it even to readers
; who know the syntax but who wish to see how Nqthm is used to
; formalize ideas that are already precisely (but informally)
; understood. We urge such readers to compare the formal defini-
; tions with the corresponding informal definitions of Chapter 4.
```

```
; We start by initializing Nqthm's data base to the GROUND-ZERO
; theory.
```

```
(BOOT-STRAP NQTHM)
```

```
; SECTION: Conventions Concerning Characters
```

```
; In the text we imagine that we have integers, characters and
; sequences of characters as our atoms. We then build token trees
; as sequences of these atoms and other token trees. Thus, the
; sequence consisting of 65, 66, and 67 is uniquely recognized as
; being a sequence of integers, while the sequence consisting of
; the characters A, B, and C, is recognized as being a word.
```

```
; But in this formalization, we do not have characters and in fact
; we use their ASCII codes instead. But the ASCII codes are them-
; selves integers. Thus, the sequence containing 65, 66, and 67 is
; ambiguous as a token tree: is it a tree containing three integers
; or is it a tree containing the word ABC? Fortunately, characters
; never enter token trees except as the elements of words and the
; various tokens. Therefore, in this formalization we use LITATOMs
; to represent the words and the tokens. That is, where the text
; represents the word ABC as a sequence consisting of the char-
; acters A, B, and C, the formalization will represent it as a
; LITATOM obtained by PACKing the ASCII codes for A, B, and C
; (around a 0). Similarly, the backquote token in the text is the
; sequence consisting of the backquote character, but here it is
; the LITATOM obtained by packing the ASCII code for that character
; (around a 0).
```

```

; We first make it convenient to refer to the ASCII character code
; of all the printable characters. We will invent a name for each
; character, e.g., UPPER-A, LOWER-B, OPEN-PAREN, and define (ASCII
; name) to be the ASCII integer code for the named character. Thus
; (ASCII 'UPPER-A) will be 65. We will also define ASCII so that
; when given a list of names it returns the list of ASCII codes for
; the characters named in the list. Thus, (ASCII '(UPPER-A
; LOWER-A)) will be '(65 97).

```

```

; The names and their ASCII codes are given in the following
; association list:

```

```

(DEFN ASCII-TABLE NIL
  '((TAB . 9)
    (NEWLINE . 10)
    (PAGE . 12)
    (SPACE . 32)
    (RUBOUT . 127)
    (EXCLAMATION-POINT . 33) ; !
    (DOUBLE-QUOTE . 34) ; "
    (NUMBER-SIGN . 35) ; #
    (DOLLAR-SIGN . 36) ; $
    (PERCENT-SIGN . 37) ; %
    (AMPERSAND . 38) ; &
    (SINGLE-QUOTE . 39) ; '
    (OPEN-PAREN . 40) ; (
    (CLOSE-PAREN . 41) ; )
    (ASTERISK . 42) ; *
    (PLUS-SIGN . 43) ; +
    (COMMA . 44) ; ,
    (MINUS-SIGN . 45) ; -
    (DOT . 46) ; .
    (SLASH . 47) ; /
    (DIGIT-ZERO . 48) ; 0
    (DIGIT-ONE . 49) ; 1
    (DIGIT-TWO . 50) ; 2
    (DIGIT-THREE . 51) ; 3
    (DIGIT-FOUR . 52) ; 4
    (DIGIT-FIVE . 53) ; 5
    (DIGIT-SIX . 54) ; 6
    (DIGIT-SEVEN . 55) ; 7
    (DIGIT-EIGHT . 56) ; 8
    (DIGIT-NINE . 57) ; 9
    (COLON . 58) ; :
    (SEMICOLON . 59) ; ;
    (LESS-THAN-SIGN . 60) ; <
    (EQUAL-SIGN . 61) ; =
    (GREATER-THAN-SIGN . 62) ; >
    (QUESTION-MARK . 63) ; ?
    (AT-SIGN . 64) ; @
    (UPPER-A . 65) ; A
    (UPPER-B . 66) ; B
    (UPPER-C . 67) ; C
    (UPPER-D . 68) ; D

```

(UPPER-E	. 69)	; E
(UPPER-F	. 70)	; F
(UPPER-G	. 71)	; G
(UPPER-H	. 72)	; H
(UPPER-I	. 73)	; I
(UPPER-J	. 74)	; J
(UPPER-K	. 75)	; K
(UPPER-L	. 76)	; L
(UPPER-M	. 77)	; M
(UPPER-N	. 78)	; N
(UPPER-O	. 79)	; O
(UPPER-P	. 80)	; P
(UPPER-Q	. 81)	; Q
(UPPER-R	. 82)	; R
(UPPER-S	. 83)	; S
(UPPER-T	. 84)	; T
(UPPER-U	. 85)	; U
(UPPER-V	. 86)	; V
(UPPER-W	. 87)	; W
(UPPER-X	. 88)	; X
(UPPER-Y	. 89)	; Y
(UPPER-Z	. 90)	; Z
(OPEN-BRACKET	. 91)	; [
(BACKSLASH	. 92)	; \
(CLOSE-BRACKET	. 93)	;]
(UPARROW	. 94)	; ^
(UNDERSCORE	. 95)	; _
(BACKQUOTE	. 96)	; `
(LOWER-A	. 97)	; a
(LOWER-B	. 98)	; b
(LOWER-C	. 99)	; c
(LOWER-D	. 100)	; d
(LOWER-E	. 101)	; e
(LOWER-F	. 102)	; f
(LOWER-G	. 103)	; g
(LOWER-H	. 104)	; h
(LOWER-I	. 105)	; i
(LOWER-J	. 106)	; j
(LOWER-K	. 107)	; k
(LOWER-L	. 108)	; l
(LOWER-M	. 109)	; m
(LOWER-N	. 110)	; n
(LOWER-O	. 111)	; o
(LOWER-P	. 112)	; p
(LOWER-Q	. 113)	; q
(LOWER-R	. 114)	; r
(LOWER-S	. 115)	; s
(LOWER-T	. 116)	; t
(LOWER-U	. 117)	; u
(LOWER-V	. 118)	; v
(LOWER-W	. 119)	; w
(LOWER-X	. 120)	; x
(LOWER-Y	. 121)	; y

```

      (LOWER-Z      . 122)      ; z
      (OPEN-BRACE   . 123)      ; {
      (VERTICAL-BAR . 124)      ; |
      (CLOSE-BRACE  . 125)      ; }
      (TILDE        . 126)))    ; ~
(DEFN ASCII-LIST (LST)
  (IF (NLISTP LST)
      NIL
      (CONS (CDR (ASSOC (CAR LST) (ASCII-TABLE)))
              (ASCII-LIST (CDR LST)))))
(DEFN ASCII (X)
  (IF (LITATOM X)
      (CDR (ASSOC X (ASCII-TABLE)))
      (ASCII-LIST X)))

```

```

; So now we can write (ASCII 'UPPER-A) for 65 and
; (ASCII '(UPPER-A LOWER-A)) for '(65 97).

```

```

; SECTION: Token Trees

```

```

(DEFN UPPER-DIGITS NIL
  (ASCII '(DIGIT-ZERO
            DIGIT-ONE
            DIGIT-TWO
            DIGIT-THREE
            DIGIT-FOUR
            DIGIT-FIVE
            DIGIT-SIX
            DIGIT-SEVEN
            DIGIT-EIGHT
            DIGIT-NINE
            UPPER-A
            UPPER-B
            UPPER-C
            UPPER-D
            UPPER-E
            UPPER-F)))

```

```

; We define (CADRN n LST) to be equivalent to (CADD...DR LST) where
; the number of D's is n. Thus, (CADRN 2 LST) is (CADDR LST). An-
; other way to think about CADRN is that it returns the Nth element
; of LST using 0-based enumeration, e.g., (CADRN 3 '(A B C D E F))
; is 'D.

```

```

(DEFN CDRN (N LST)
  (IF (ZEROP N)
      LST
      (CDRN (SUB1 N) (CDR LST))))
(DEFN CADRN (N LST) (CAR (CDRN N LST)))

```

```

; It is also convenient to have

```

```

(DEFN LIST1 (X) (CONS X NIL))
(DEFN LIST2 (X Y) (CONS X (LIST1 Y)))
(DEFN LIST3 (X Y Z) (CONS X (LIST2 Y Z)))

```

```

; because we are eschewing the use of the abbreviation LIST.
(DEFN FIRST-N (N LST)
  (IF (ZEROP N)
      NIL
      (CONS (CAR LST)
              (FIRST-N (SUB1 N) (CDR LST))))))
(DEFN BASE-N-DIGIT-CHARACTER (N C)
  (AND (LEQ N 16)
        (MEMBER C (FIRST-N N (UPPER-DIGITS)))))
(DEFN POSITION (X LST)
  (IF (NLISTP LST)
      0
      (IF (EQUAL X (CAR LST))
          0
          (ADD1 (POSITION X (CDR LST))))))
(DEFN BASE-N-DIGIT-VALUE (C)
  (POSITION C (UPPER-DIGITS)))
(DEFN ALL-BASE-N-DIGIT-CHARACTERS (N LST)
  (IF (NLISTP LST)
      T
      (AND (BASE-N-DIGIT-CHARACTER N (CAR LST))
            (ALL-BASE-N-DIGIT-CHARACTERS N (CDR LST)))))
(DEFN BASE-N-DIGIT-SEQUENCE (N LST)
  (AND (LISTP LST)
        (ALL-BASE-N-DIGIT-CHARACTERS N LST)))
(DEFN OPTIONALLY-SIGNED-BASE-N-DIGIT-SEQUENCE (N LST)
  (OR (BASE-N-DIGIT-SEQUENCE N LST)
      (AND (LISTP LST)
            (AND (OR (EQUAL (CAR LST) (ASCII 'PLUS-SIGN))
                    (EQUAL (CAR LST) (ASCII 'MINUS-SIGN)))
                (BASE-N-DIGIT-SEQUENCE N (CDR LST))))))
(DEFN LENGTH (LST)
  (IF (NLISTP LST)
      0
      (ADD1 (LENGTH (CDR LST)))))
(DEFN EXP (I J)
  (IF (ZEROP J)
      1
      (TIMES I (EXP I (SUB1 J)))))
(DEFN BASE-N-VALUE (N LST)
  (IF (NLISTP LST)
      0
      (PLUS (TIMES (BASE-N-DIGIT-VALUE (CAR LST))
                  (EXP N (LENGTH (CDR LST)))))
            (BASE-N-VALUE N (CDR LST)))))
(DEFN NUMERATOR-SEQUENCE (LST)
  (IF (NLISTP LST)
      NIL
      (IF (EQUAL (CAR LST) (ASCII 'SLASH))
          NIL
          (CONS (CAR LST)
                  (NUMERATOR-SEQUENCE (CDR LST))))))

```

```

(DEFN DENOMINATOR-SEQUENCE (LST)
  (IF (NLISTP LST)
      NIL
      (IF (EQUAL (CAR LST) (ASCII 'SLASH))
          (CDR LST)
          (DENOMINATOR-SEQUENCE (CDR LST)))))
(DEFN BASE-N-SIGNED-VALUE (N LST)
  (IF (EQUAL (CAR LST) (ASCII 'MINUS-SIGN))
      (MINUS (BASE-N-VALUE N (CDR LST)))
      (IF (EQUAL (CAR LST) (ASCII 'PLUS-SIGN))
          (BASE-N-VALUE N (CDR LST))
          (BASE-N-VALUE N LST))))
(DEFN NUMBER-SIGN-SEQUENCE (LST)
  (AND (GEQ (LENGTH LST) 3)
       (EQUAL (CAR LST) (ASCII 'NUMBER-SIGN))
       (MEMBER (CADRN 1 LST)
                (ASCII '(UPPER-B UPPER-O UPPER-X)))
       (OPTIONALLY-SIGNED-BASE-N-DIGIT-SEQUENCE
        (IF (EQUAL (CADRN 1 LST) (ASCII 'UPPER-B))
            2
            (IF (EQUAL (CADRN 1 LST) (ASCII 'UPPER-O))
                8
                16)))
        (CDR (CDR LST)))))
(DEFN LAST (LST)
  (IF (NLISTP LST)
      LST
      (IF (NLISTP (CDR LST))
          LST
          (LAST (CDR LST)))))
(DEFN ALL-BUT-LAST (LST)
  (IF (NLISTP LST)
      NIL
      (IF (NLISTP (CDR LST))
          NIL
          (CONS (CAR LST)
                 (ALL-BUT-LAST (CDR LST)))))
(DEFN NUMERIC-SEQUENCE (LST)
  (OR (OPTIONALLY-SIGNED-BASE-N-DIGIT-SEQUENCE 10 LST)
      (OR (AND (EQUAL (CAR (LAST LST)) (ASCII 'DOT))
              (OPTIONALLY-SIGNED-BASE-N-DIGIT-SEQUENCE
               10
               (ALL-BUT-LAST LST)))
          (NUMBER-SIGN-SEQUENCE LST))))
  (DEFN NUMERIC-VALUE (LST)
    (IF (OPTIONALLY-SIGNED-BASE-N-DIGIT-SEQUENCE 10 LST)
        (BASE-N-SIGNED-VALUE 10 LST)
        (IF (EQUAL (CAR (LAST LST)) (ASCII 'DOT))
            (BASE-N-SIGNED-VALUE 10 (ALL-BUT-LAST LST))
            (BASE-N-SIGNED-VALUE
             (IF (EQUAL (CADRN 1 LST) (ASCII 'UPPER-B))
                 2
                 (IF (EQUAL (CADRN 1 LST) (ASCII 'UPPER-O))
                     8
                     16)))
             (CDR (CDR LST))))))

```

```

      8
      16))
      (CDR (CDR LST))))))
(DEFN SINGLE-QUOTE-TOKEN () (PACK (CONS (ASCII 'SINGLE-QUOTE) 0)))
(DEFN BACKQUOTE-TOKEN   () (PACK (CONS (ASCII 'BACKQUOTE) 0)))
(DEFN DOT-TOKEN          () (PACK (CONS (ASCII 'DOT) 0)))
(DEFN COMMA-TOKEN        () (PACK (CONS (ASCII 'COMMA) 0)))
(DEFN COMMA-AT-SIGN-TOKEN () (PACK (CONS (ASCII 'COMMA)
                                         (CONS (ASCII 'AT-SIGN) 0))))
(DEFN COMMA-DOT-TOKEN    () (PACK (CONS (ASCII 'COMMA)
                                         (CONS (ASCII 'DOT) 0))))
(DEFN WORD-CHARACTERS ()
  (ASCII
    '(UPPER-A UPPER-B UPPER-C UPPER-D UPPER-E UPPER-F UPPER-G
      UPPER-H UPPER-I UPPER-J UPPER-K UPPER-L UPPER-M UPPER-N
      UPPER-O UPPER-P UPPER-Q UPPER-R UPPER-S UPPER-T UPPER-U
      UPPER-V UPPER-W UPPER-X UPPER-Y UPPER-Z
      DIGIT-ZERO DIGIT-ONE DIGIT-TWO DIGIT-THREE DIGIT-FOUR
      DIGIT-FIVE DIGIT-SIX DIGIT-SEVEN DIGIT-EIGHT DIGIT-NINE
      DOLLAR-SIGN UPARROW AMPERSAND ASTERISK UNDERSCORE
      MINUS-SIGN PLUS-SIGN EQUAL-SIGN TILDE OPEN-BRACE CLOSE-BRACE
      QUESTION-MARK LESS-THAN-SIGN GREATER-THAN-SIGN)))
(DEFN SUBSETP (X Y)
  (IF (NLISTP X)
      T
      (IF (MEMBER (CAR X) Y)
          (SUBSETP (CDR X) Y)
          F)))
(DEFN WORD (S)
  (AND (LITATOM S)
        (OR (NUMERIC-SEQUENCE (UNPACK S))
            (AND (LISTP (UNPACK S))
                  (SUBSETP (UNPACK S) (WORD-CHARACTERS))))))

; It is convenient to be able to recognize those words that are
; numeric sequences and to talk about their numeric values, without
; having to think about unpacking them. So we define NUMERIC-WORD
; and NUMERIC-WORD-VALUE and use them below where the text would
; have us use ``numeric sequence`` and ``numeric value.``
(DEFN NUMERIC-WORD (S)
  (AND (LITATOM S)
        (NUMERIC-SEQUENCE (UNPACK S))))
(DEFN NUMERIC-WORD-VALUE (S)
  (NUMERIC-VALUE (UNPACK S)))
(DEFN INTEGERP (X)
  (OR (NUMBERP X)
      (NEGATIVEP X)))
(DEFN SPECIAL-TOKEN (X)
  (OR (EQUAL X (SINGLE-QUOTE-TOKEN))
      (OR (EQUAL X (BACKQUOTE-TOKEN))
          (OR (EQUAL X (COMMA-TOKEN))
              (OR (EQUAL X (COMMA-AT-SIGN-TOKEN))
                  (EQUAL X (COMMA-DOT-TOKEN)))))))

```



```

; The following function can be used as follows.  If we test
; (EQLEN X 3) then we know that lst is of the form (x1 x2 x3).
(DEFN EQLEN (LST N)
  (IF (ZEROP N)
      (EQUAL LST NIL)
      (IF (NLISTP LST)
          F
          (EQLEN (CDR LST) (SUB1 N)))))
(DEFN DOTTED-PAIR (X)
  (AND (EQLEN X 3)
       (EQUAL (CADR N 1 X) (DOT-TOKEN))))
(DEFN DOTTED-S-EXPRESSION (X)
  (IF (NLISTP X)
      F
      (IF (DOTTED-PAIR X)
          T
          (DOTTED-S-EXPRESSION (CDR X)))))
(DEFN SINGLETON (X)
  (EQLEN X 1))

; We use the following lemma to make the CDRNs go away during the
; termination proof for TOKEN-TREE.

(PROVE-LEMMA CDRN-EXPANDER (REWRITE)
  (EQUAL (CDRN (ADD1 N) X) (CDRN N (CDR X))))

; Now we define token trees formally.  We use the logic's LISTPs,
; together with the integers, words and tokens (the last two being
; LITATOMs), to represent token trees.  We are faithful to the
; text's style of representing dotted token trees by lists whose
; second-to-last elements are the dot token.  That is, we don't
; represent such trees by dotted pairs in the logic.  (This would
; not work because the token tree (A B . (C D E)) is legitimate and
; is different from (A B C D E).  The text permits such token trees
; since they may be typed.  They readmacro-expand to the same
; s-expression.)  However, we do not define ``token tree'' in quite
; the same style as the text, though the result is the same.  The
; text recognizes undotted and dotted token trees ``from the
; outside'' -- e.g., an undotted token tree is a nonempty sequence
; of token trees and a dotted one is similar but has a dot as its
; second-to-last element.  To define token trees formally this way
; we would need to use mutual recursion.  Rather than do that, we
; just recurse down dotted and undotted lists and catch the dot
; when we come to it.
(DEFN TOKEN-TREE (X)
  (IF (NLISTP X)
      (OR (INTEGERP X)
          (WORD X))
      (IF (AND (SPECIAL-TOKEN (CAR X))
               (EQUAL (LENGTH X) 2))
          (TOKEN-TREE (CADR N 1 X))
          (IF (DOTTED-PAIR X)
              (AND (TOKEN-TREE (CAR X))
                   (TOKEN-TREE (CDR X))

```

```

      (TOKEN-TREE (CADRN 2 X)))
    (IF (SINGLETON X)
        (TOKEN-TREE (CAR X))
        (AND (TOKEN-TREE (CAR X))
              (TOKEN-TREE (CDR X)))))))))

; We define a few functions to make token tree manipulation and
; recognition easier.
(DEFN SPECIAL-TOKEN-TREE (X)
  (SPECIAL-TOKEN (CAR X)))
(DEFN SINGLE-QUOTE-TOKEN-TREE (X)
  (EQUAL (CAR X) (SINGLE-QUOTE-TOKEN)))
(DEFN BACKQUOTE-TOKEN-TREE (X)
  (EQUAL (CAR X) (BACKQUOTE-TOKEN)))
(DEFN COMMA-ESCAPE-TOKEN-TREE (X)
  (EQUAL (CAR X) (COMMA-TOKEN)))
(DEFN SPLICE-ESCAPE-TOKEN-TREE (X)
  (OR (EQUAL (CAR X) (COMMA-AT-SIGN-TOKEN))
      (EQUAL (CAR X) (COMMA-DOT-TOKEN))))
(DEFN CONSTITUENT (X) (CADRN 1 X))

; SECTION: Reading Token Trees

; At this point in the text, we define how to display a token tree.
; We will skip that here and instead formalize the process of pars-
; ing a token tree from a sequence of characters (i.e., from a dis-
; play of one).

; Our parser is reminiscent of Lisp's read routine with several im-
; portant exceptions. We do not deal with readmacros in our parser
; -- that is the job of readmacro-expansion (defined later). We
; also do not produce dotted pairs but rather lists containing the
; dot token. Finally, we can produce ``unreadable'' token trees
; such as parsed from (PLUS ,X B).

; The parser is called READ-TOKEN-TREE and it is implemented in two
; passes. The first pass transforms a list of ASCII character
; codes into a list of lexemes. In our case, all the lexemes are
; LITATOMs, including the numeric sequences that look like inte-
; gers, i.e., we resolve the ambiguity in ``display'' by always
; using numeric sequences. The second pass parses the lexemes into
; a tree. In fact, the parser can produce trees that are not token
; trees, such as the one produced by parsing (PLUS # B). This
; ``pseudo-token tree'' fails to be a token tree because one of the
; atoms in it that ``should'' be a word is not a word. READ-TOKEN-
; TREE therefore calls the predicate TOKEN-TREE on the parsed tree
; to determine whether the parse was successful.

```

```

; SUBSECTION: Pass I of the Reader

; We give convenient names to the lexemes for open and close
; parentheses. We could call these ``tokens'' but don't want to
; confuse them with the tokens of the text. The lexemes for the
; tokens, e.g., the dot lexeme, will be the tokens themselves,
; e.g., the value of (DOT-TOKEN).
(DEFN OPEN-PAREN  () (PACK (CONS (ASCII 'OPEN-PAREN) 0)))
(DEFN CLOSE-PAREN () (PACK (CONS (ASCII 'CLOSE-PAREN) 0)))

; The following function scans past characters until it has passed
; the first newline character. It is used to scan past semicolon
; comment. It returns the stream starting just after that newline.
; If no newline is found, it returns the empty stream. The ques-
; tion then arises: was the comment that started this scan well-
; formed or not? It is missing its final newline. It turns out
; that Common Lisp's answer to this is that it is ok. That is, a
; file can end without there being a final newline on the end of a
; comment on the last line of the file. The text of Chapter 4 does
; not deal with this issue.
(DEFN SKIP-PAST-NEWLINE (STREAM)
  (IF (NLISTP STREAM)
      STREAM
      (IF (EQUAL (CAR STREAM) (ASCII 'NEWLINE))
          (CDR STREAM)
          (SKIP-PAST-NEWLINE (CDR STREAM))))))

; The lexical analyzer will sometimes recurse using this function
; to ``decrement'' the input stream. We must prove that the stream
; returned is weakly smaller (the semicolon that starts the
; comment will have already been stripped off by an explicit CDR).
; We measure size here with LENGTH instead of COUNT because COUNT
; doesn't decrease for all of the recursions (see the next function
; definition).

(PROVE-LEMMA LESSP-SKIP-PAST-NEWLINE (REWRITE)
  (NOT (LESSP (LENGTH STREAM)
              (LENGTH (SKIP-PAST-NEWLINE STREAM))))))

; The next function scans for the end of a just-opened #-comment.
; The function returns the stream just past the closing sequence of
; the comment. The variable I below counts the number of ``open''
; #| sequences we have seen. We do not stop until we see a|#
; sequence that closes that one, i.e., decrements I to 0. I is
; initially 1 because we call this function after reading the
; opening sequence. Note that this function will scan the entire
; input stream if the balancing sequence is not present. This is
; treated as an error by Common Lisp. We therefore have the
; problem of signaling this error. If we return the empty stream,
; it is indistinguishable from a well-formed comment that ends at
; the very end of the stream. We therefore use a trick: we return
; the stream containing a single open parenthesis character. This
; character will cause the lexical analyzer to put an unbalanced

```

```
; open parenthesis lexeme as the last lexeme in the stream fed to
; the parser. That, in turn, will cause an error. We also signal
; such an error if we find too many |# sequences. A minor tech-
; nical problem arises: to insure that the lexical analyzer can
; recurse with this skipping function, we have to make sure the
; size of the stream we return is less than the one the lexical
; analyzer started with. Since the analyzer will have CDRd past
; the opening #| our adding a single open parenthesis in the place
; of those two characters will not matter -- if we measure the size
; of the stream with LENGTH! If we were to measure with COUNT, we
; would have to worry about the size of the various ASCII codes and
; about the object in the final CDR of the stream.
```

```
(DEFN SKIP-PAST-BALANCING-VERTICAL-BAR-NUMBER-SIGN (STREAM I)
  (IF (NLISTP STREAM)
    (CONS (ASCII 'OPEN-PAREN) STREAM)
    (IF (AND (EQUAL (CAR STREAM) (ASCII 'NUMBER-SIGN))
      (EQUAL (CADR 1 STREAM) (ASCII 'VERTICAL-BAR)))
      (SKIP-PAST-BALANCING-VERTICAL-BAR-NUMBER-SIGN
        (CDR 2 STREAM)
        (ADD1 I))
      (IF (AND (EQUAL (CAR STREAM) (ASCII 'VERTICAL-BAR))
        (EQUAL (CADR 1 STREAM) (ASCII 'NUMBER-SIGN)))
        (IF (EQUAL I 0)
          (CONS (ASCII 'OPEN-PAREN) (CDR 2 STREAM))
          (IF (EQUAL I 1)
            (CDR 2 STREAM)
            (SKIP-PAST-BALANCING-VERTICAL-BAR-NUMBER-SIGN
              (CDR 2 STREAM)
              (SUB1 I))))
          (SKIP-PAST-BALANCING-VERTICAL-BAR-NUMBER-SIGN
            (CDR STREAM)
            I))))))
```

```
; This is the key inductive fact about the relative LENGTHs of the
; input and output of the function above:
```

```
(PROVE-LEMMA SKIP-PAST-BALANCING-VERTICAL-BAR-NUMBER-SIGN-LEMMA
  (REWRITE)
  (NOT (LESSP
    (ADD1 (LENGTH STREAM))
    (LENGTH (SKIP-PAST-BALANCING-VERTICAL-BAR-NUMBER-SIGN
      STREAM
      I))))))
```

; However, this is the actual theorem we need to admit the lexical
; analyzer (eventually) below:

```
(PROVE-LEMMA LESSP-SKIP-PAST-BALANCING-VERTICAL-BAR-NUMBER-SIGN
  (REWRITE)
  (IMPLIES (AND (LISTP STREAM)
                (AND (EQUAL (CAR STREAM) (ASCII 'NUMBER-SIGN))
                    (EQUAL (CADR STREAM) (ASCII 'VERTICAL-BAR)))))
    (LESSP
     (LENGTH
      (SKIP-PAST-BALANCING-VERTICAL-BAR-NUMBER-SIGN
       (CDDR STREAM)
       1))
     (LENGTH STREAM))))
```

; To recognize white space we need:

```
(DEFN WHITE-SPACEP (C)
  (MEMBER C (ASCII '(SPACE TAB NEWLINE))))
```

; We will accumulate the characters in a lexeme in a list (with a 0
; at the bottom) and when we have a completed lexeme we will add it
; to a growing list of lexemes. The characters are accumulated in
; reverse order. The empty lexeme is never added to the list.

```
(DEFN REV1 (LST ANS)
  (IF (NLISTP LST)
      ANS
      (REV1 (CDR LST) (CONS (CAR LST) ANS))))
(DEFN EMIT (PNAME LST)
  (IF (EQUAL PNAME 0)
      LST
      (CONS (PACK (REV1 PNAME 0)) LST)))
```

; The following function maps lower case alphabetic characters into
; their upper case counterparts. Thus, (UPCASE (ASCII 'LOWER-A))
; is (ASCII 'UPPER-A), etc. UPCASE is the identity function on
; characters other than LOWER-A through LOWER-Z.

```
(DEFN UPCASE (C)
  (IF (AND (LEQ (ASCII 'LOWER-A) C)
          (LEQ C (ASCII 'LOWER-Z)))
      (DIFFERENCE C (DIFFERENCE (ASCII 'LOWER-A) (ASCII 'UPPER-A)))
      C))
```

; Here then is the lexical analyzer, pass I of the parser. The
; function returns a list of lexemes parsed from stream, which is a
; list of ASCII codes. PNAME is the lexeme currently being
; assembled. It is 0 when ``empty.''

```
(DEFN LEXEMES (STREAM PNAME)
  (IF (NLISTP STREAM)
      (EMIT PNAME NIL)
      (IF (EQUAL (CAR STREAM) (ASCII 'SEMICOLON))
```

; A semicolon terminates the accumulating lexeme and causes us to
; skip to the next line.

```

      (EMIT PNAME
        (LEXEMES (SKIP-PAST-NEWLINE (CDR STREAM))
          0))

; Backquote, single quote, and the two parens terminate the lexeme
; and also emit the corresponding token or lexeme.

      (IF (MEMBER (CAR STREAM)
        (ASCII '(BACKQUOTE
          SINGLE-QUOTE
          OPEN-PAREN
          CLOSE-PAREN))))
      (EMIT PNAME
        (EMIT (CONS (CAR STREAM) 0)
          (LEXEMES (CDR STREAM) 0)))

      (IF (EQUAL (CAR STREAM) (ASCII 'COMMA))

; Comma terminates the lexeme but emits the comma, comma at-sign,
; or comma dot token, depending on the next character.

        (IF (OR (EQUAL (CADRN 1 STREAM) (ASCII 'AT-SIGN))
          (EQUAL (CADRN 1 STREAM) (ASCII 'DOT)))
          (EMIT PNAME
            (EMIT (CONS (CADRN 1 STREAM) (CONS (CAR STREAM) 0))
              (LEXEMES (CDR (CDR STREAM)) 0)))
          (EMIT PNAME
            (EMIT (CONS (CAR STREAM) 0)
              (LEXEMES (CDR STREAM) 0)))))

      (IF (WHITE-SPACEP (CAR STREAM))

; White space terminates the lexeme and emits nothing.

        (EMIT PNAME (LEXEMES (CDR STREAM) 0))

      (IF (AND (EQUAL PNAME 0)
        (AND (EQUAL (CAR STREAM) (ASCII 'NUMBER-SIGN))
          (EQUAL (CADRN 1 STREAM) (ASCII 'VERTICAL-BAR)))))

; Finally, number-sign does not terminate the lexeme, but if it
; occurs while no lexeme is being accumulated and is immediately
; followed by vertical bar, it signals the beginning of a comment.

      (LEXEMES (SKIP-PAST-BALANCING-VERTICAL-BAR-NUMBER-SIGN
        (CDRN 2 STREAM)
        1)
        0)

; Otherwise, we just accumulate the character onto the lexeme being
; assembled. This case includes DOT, which is not a break char-
; acter and thus must appear in isolation to be parsed as a dot
; token.

```

```

      (LEXEMES (CDR STREAM)
        (CONS (UPCASE (CAR STREAM)) PNAME))))))
    ((LESSP (LENGTH STREAM)))

; SUBSECTION: Pass II of the Reader

; We now develop the parser, which maps a list of lexemes into a
; tree -- or returns F if the list does not parse. One example of
; a list of lexemes that doesn't parse is one containing unbalanced
; parentheses. Another example is one that contains more lexemes
; than necessary to describe exactly one tree. That is, our parser
; balks if there are lexemes left over when one tree has been
; parsed.

; The formalization below is just a simple push-down stack auto-
; maton. It scans the lexemes one at a time from the right all the
; way to the end of the list. It has a stack of trees it is build-
; ing up. When it sees an open parenthesis, it pushes an empty
; frame onto the stack. When it sees a normal lexeme, like the
; word PLUS or the integer 123, it accumulates this onto the right
; end of the top-most frame. When it sees a close parenthesis, it
; pops the top frame and accumulates it onto the right end of the
; frame below. We code it so that if any of these stack operations
; is ill-formed, e.g., we pop an empty stack, the result is F.
; Furthermore, we arrange for an F stack to be propagated, i.e.,
; pushing something onto F produces F. Thus, if a parsing error
; arises early in the scan of the lexemes, e.g., an unbalanced
; close parenthesis is seen, the stack becomes F and even though
; the scan continues until the last lexeme has been processed, the
; F is preserved as the signal that an error occurred.

; When all the lexemes have been processed we inspect the stack and
; verify that it contains a single frame. If not, then we return
; F. (If the stack contains less than one frame, then the stack is
; in fact F and an error was detected earlier. If the stack
; contains more than one frame, we have unbalanced open paren-
; theses.)

; To make this work we have to start the stack as though we are
; accumulating a list, i.e., with one empty frame on the stack.
; Then when we are done we have to check that the list we accumu-
; lated has exactly one element. If it has none, the input was
; empty, which is an error. If it has more than one, the input
; contained more than one tree and that is an error.

; To handle the special tokens single-quote, backquote, comma,
; comma-at-sign, and comma-dot, we generalize the stack slightly so
; that when these tokens are read they push a new frame that con-
; tains the token rather than the empty list. Then we continue
; parsing. When the next tree is assembled it is ``accumulated
; onto the right end of the frame below,'' where we actually take
; into account the special tokens that might mark the frame below.
; For example, if x is to be accumulated onto the frame below, and

```

```

; the frame below contains the single-quote token, then we create
; the single-quote tree (' x) and recursively accumulate it onto
; the frame below that.

; Finally, the dot token is afforded no such special handling
; except that when a list is accumulated onto another (or returned
; as the answer) we verify that if it contains the dot token as an
; element then the token occurs as the next-to-last element.

; Of course, this whole process is much simpler than the Lisp
; reader because we are not implementing readmacros here, nor do we
; have to do the checks for ``readability.'' We just parse a tree
; and return it for the rest of this system to inspect.

; So here are the functions implementing our stacks
(DEFN TOP-PSTK (STACK) (IF (LISTP STACK) (CAR STACK) F))
(DEFN POP-PSTK (STACK) (IF (LISTP STACK) (CDR STACK) F))
(DEFN PUSH-PSTK (X STACK) (IF (EQUAL STACK F) F (CONS X STACK)))
(DEFN EMPTY-PSTK (X) (NLISTP X))

; Here is the predicate that verifies that a dotted tree is prop-
; erly formed. We permit x to be the dot token, even though that
; is not a token tree, so that it can be accumulated just like
; other atoms. We will filter out the isolated dot at the top. So
; what we want to check here is that if X is a list and the dot
; token appears as an element then it appears in the next-to-last
; position and there is at least one element before it. The vari-
; able I below just counts the number of elements we've scanned
; past.
(DEFN DOT-CRITERION (I X)
  (IF (NLISTP X)
      T
      (IF (EQUAL (CAR X) (DOT-TOKEN))
          (AND (NOT (ZEROP I))
                (AND (LISTP (CDR X))
                      (NLISTP (CDR (CDR X)))))
          (DOT-CRITERION (ADD1 I) (CDR X)))))

; Here is how we accumulate X ``onto the frame below,'' where the
; ``frame below'' is the top of the stack passed into this func-
; tion. We check first that X satisfies the dot token criterion
; and cause an error if it doesn't. Then we handle the special
; tokens.
(DEFN ADD-ELEMENT-TO-TOP (X STACK)
  (IF (EMPTY-PSTK STACK)
      F
      (IF (NOT (DOT-CRITERION 0 X))
          F
          (IF (SPECIAL-TOKEN (TOP-PSTK STACK))
              (ADD-ELEMENT-TO-TOP (LIST2 (TOP-PSTK STACK) X)
                                   (POP-PSTK STACK))
              (PUSH-PSTK (APPEND (TOP-PSTK STACK) (LIST1 X))
                          (POP-PSTK STACK)))))))

```


; When we are all done, the following function is used to verify
 ; that exactly one whole object was constructed and that it is not
 ; an isolated dot-token.

```
(DEFN STOP (STACK)
  (IF (AND (EQLEN STACK 1)
    (AND (EQLEN (TOP-PSTK STACK) 1)
      (NOT (EQUAL (CAR (TOP-PSTK STACK)) (DOT-TOKEN))))))
    (CAR (TOP-PSTK STACK))
    F))
```

; Finally, here is the parser. The proper top-level call of this
 ; function is (PARSE LEXEMES (CONS NIL NIL)). It returns either a
 ; tree or F. F indicates that the lexemes either did not parse in-
 ; to a complete tree or parsed into more than one. The tree may be
 ; ``unreadable.''

```
(DEFN PARSE (LEXEMES STACK)
  (IF (NLISTP LEXEMES)
    (STOP STACK)
    (IF (EQUAL (CAR LEXEMES) (OPEN-PAREN))
      (PARSE (CDR LEXEMES) (PUSH-PSTK NIL STACK))
      (IF (SPECIAL-TOKEN (CAR LEXEMES))
        (PARSE (CDR LEXEMES) (PUSH-PSTK (CAR LEXEMES) STACK))
        (IF (EQUAL (CAR LEXEMES) (CLOSE-PAREN))
          (PARSE (CDR LEXEMES)
            (ADD-ELEMENT-TO-TOP (TOP-PSTK STACK)
              (POP-PSTK STACK)))
          (PARSE (CDR LEXEMES)
            (ADD-ELEMENT-TO-TOP (CAR LEXEMES) STACK)))))))
```

; SUBSECTION: Putting the Two Passes Together

; As noted, the parser may not produce a token tree because the
 ; lexical analyzer can produce non-word, non-token lexemes. For
 ; example, the stream (ASCII '(SPACE NUMBER-SIGN SPACE)) parses as
 ; the ``word'' we might display as # and which is logically repre-
 ; sented by the literal atom (PACK (CONS 35 0)). But this is not a
 ; word, technically, because the number sign character is not among
 ; the word characters. Therefore, after we have parsed the lexemes
 ; we check that the result is a token tree and return F if it is
 ; not. Thus, this function may return F because
 ; (a) the lexical analyzer found fault with the character stream
 ; (e.g., an unbalanced #-comment)
 ; (b) the parser found fault with the lexeme stream (e.g., an un-
 ; balanced open parenthesis), or
 ; (c) the final tree failed to be a token tree.

```
(DEFN READ-TOKEN-TREE (STREAM)
  (IF (TOKEN-TREE (PARSE (LEXEMES STREAM 0) (CONS NIL NIL)))
    (PARSE (LEXEMES STREAM 0) (CONS NIL NIL))
    F))
```

```
; SECTION: Readable Token Trees and S-Expressions
```

```
; We now resume our formalization of the text. The definition of
; ``readable from depth'' N assumes X is a token tree or F. We
; allow F as a possible input only because that is the error signal
; presented by READ-TOKEN-TREE. F is not a token tree -- the only
; atomic token trees are integers and words. Our formalization of
; ``readable'' announces that F is not readable, passing the
; ``error'' up.
```

```
(DEFN READABLE (X N)
  (IF (NLISTP X)
      (NOT (EQUAL X F))
      (IF (SINGLE-QUOTE-TOKEN-TREE X)
          (READABLE (CONSTITUENT X) N)
          (IF (BACKQUOTE-TOKEN-TREE X)
              (AND (READABLE (CONSTITUENT X) (ADD1 N))
                    (NOT (SPLICE-ESCAPE-TOKEN-TREE (CONSTITUENT X))))
              (IF (OR (COMMA-ESCAPE-TOKEN-TREE X)
                      (SPLICE-ESCAPE-TOKEN-TREE X))
                  (AND (LESSP 0 N)
                        (READABLE (CONSTITUENT X) (SUB1 N)))
                  (IF (DOTTED-PAIR X)
                      (AND (READABLE (CAR X) N)
                           (AND (READABLE (CADRN 2 X) N)
                                (NOT (SPLICE-ESCAPE-TOKEN-TREE (CADRN 2 X))))))
                      (IF (SINGLETON X)
                          (READABLE (CAR X) N)
                          (AND (READABLE (CAR X) N)
                               (READABLE (CDR X) N))))))))))
```

```
; Like our formalization of ``readable,'' our formalization of ``s-
; expression'' allows its argument to be either a token tree or F
; and returns F on F. Thus, if you apply S-EXPRESSION to the out-
; put of READ-TOKEN-TREE you will get F if the read ``caused an
; error.'' In actual use, we only apply S-EXPRESSION to the output
; of READMACRO-EXPANSION and we only apply that to READABLE token
; trees. Thus, this aspect of our formalization of s-expressions
; is irrelevant. We preserve it because it is sometimes nice,
; while testing these definitions, to run S-EXPRESSION on the
; output of the reader.
```

```
(DEFN S-EXPRESSION (X)
  (IF (NLISTP X)
      (IF (EQUAL X F)
          F
          (NOT (NUMERIC-WORD X)))
      (IF (SPECIAL-TOKEN-TREE X)
          F
          (IF (DOTTED-PAIR X)
              (AND (S-EXPRESSION (CAR X)) (S-EXPRESSION (CADRN 2 X)))
              (IF (SINGLETON X)
                  (S-EXPRESSION (CAR X))
                  (AND (S-EXPRESSION (CAR X))
                       (S-EXPRESSION (CDR X))))))))))
```

```
; SECTION: Backquote Expansion
```

```
; At this point in the text we give the definition of readmacro-
; expansion. However, it is presented without first defining the
; notion of backquote expansion. We therefore skip ahead in the
; text and formalize backquote expansion now so we can then present
; readmacro-expansion.
```

```
; Of course, backquote expansion and backquote-list expansion are
; mutually recursive. If FLG below is T we are defining backquote
; expansion. Otherwise, we are defining backquote-list expansion.
(DEFN BACKQUOTE-EXPANSION (FLG X)
```

```
  (IF FLG
    (IF (NLISTP X)
      (LIST2 'QUOTE X)
      (IF (OR (COMMA-ESCAPE-TOKEN-TREE X)
              (SPLICE-ESCAPE-TOKEN-TREE X))
        (CONSTITUENT X)
        (BACKQUOTE-EXPANSION F X)))
    (IF (NLISTP X)
      'IMPOSSIBLE-IF-X-IS-A-DOTTED-OR-UNDOTTED-TOKEN-TREE
      (LIST3 (IF (SPLICE-ESCAPE-TOKEN-TREE (CAR X))
                  'APPEND
                  'CONS)
              (BACKQUOTE-EXPANSION T (CAR X))
              (IF (SINGLETON X)
                (LIST2 'QUOTE 'NIL)
                (IF (DOTTED-PAIR X)
                  (BACKQUOTE-EXPANSION T (CADRN 2 X))
                  (BACKQUOTE-EXPANSION F (CDR X)))))))
    ((ORD-LESSP (CONS (ADD1 (COUNT X)) (IF FLG 1 0))))))
```

```
; SECTION: Readmacro Expansion
```

```
; Our definition of readmacro-expansion differs from the text in
; that we do all four passes at once. It is so easy in English to
; say ``replace every ...'' and its formalization requires a recur-
; sive sweep through the tree. Even if we had higher order func-
; tions (or used V&C$) we would spend about as much space defining
; the generic sweep as we do below just doing it.
```

```
(DEFN READMACRO-EXPANSION (X)
  (IF (NLISTP X)
    (IF (NUMERIC-WORD X)
      (NUMERIC-WORD-VALUE X)
      X)
    (IF (SINGLE-QUOTE-TOKEN-TREE X)
      (LIST2 'QUOTE (READMACRO-EXPANSION (CONSTITUENT X)))
      (IF (BACKQUOTE-TOKEN-TREE X)
        (BACKQUOTE-EXPANSION T
          (READMACRO-EXPANSION (CONSTITUENT X)))
        (IF (DOTTED-PAIR X)
          (IF (OR (AND (NLISTP (READMACRO-EXPANSION (CADRN 2 X)))
```

```

        (NOT (EQUAL (READMACRO-EXPANSION (CADRN 2 X))
                    'NIL)))
      (SPECIAL-TOKEN-TREE
       (READMACRO-EXPANSION (CADRN 2 X))))
    (LIST3 (READMACRO-EXPANSION (CAR X))
           (DOT-TOKEN)
           (READMACRO-EXPANSION (CADRN 2 X)))
    (CONS (READMACRO-EXPANSION (CAR X))
          (READMACRO-EXPANSION (CADRN 2 X))))
  (IF (SINGLETON X)
      (LIST1 (READMACRO-EXPANSION (CAR X)))
      (CONS (READMACRO-EXPANSION (CAR X))
            (READMACRO-EXPANSION (CDR X)))))))))

; SECTION: Some Common Lisp

#|

; This entire section is commented out. It is here only as a con-
; venience for those wishing to test the token tree reader and
; readmacro-expansion. The following function reads a token tree
; from a stream of ASCII character codes. If it did not parse or
; is ``unreadable,`` suitable messages are returned. Otherwise,
; the token tree is readmacro-expanded. If an s-expression does
; not result, a suitable message is returned. THIS ERROR MESSAGE
; SHOULD NEVER HAPPEN because the readmacro-expansion of a readable
; token tree supposedly always produces an s-expression! If no
; errors are reported, the function returns the resulting s-
; expression. Thus, this function is essentially a formalization
; of the the Lisp read routine.
(DEFN TEST-READER (STREAM)
  (LET ((X (READ-TOKEN-TREE STREAM)))
    (COND ((NOT X) 'DID-NOT-PARSE)
          ((NOT (READABLE X 0)) 'NOT-READABLE-FROM-0)
          (T (LET ((Y (READMACRO-EXPANSION X)))
                (COND
                 ((S-EXPRESSION Y) Y)
                 (T (LIST
                     'READMACRO-EXPANSION-PRODUCED-NON-S-EXPRESSION
                     Y))))))))))

; The following defines a Lisp routine, not a function in the
; logic. The routine's name is test-reader and it is just a
; convenient interface to the logical function defined above. You
; give it a Lisp string and it converts it into a list of ASCII
; codes. This just lets us type things like (test-reader
; "(PLUS X Y)") to Common Lisp -- not to R-LOOP -- to execute the
; logic's TEST-READER.

(defun ascii (string)
  (mapcar (function char-code)
          (coerce string 'list)))

```

```

(defun test-reader (string)
  (*1*test-reader (ascii string)))

; We now return to the main flow of this development of the
; extended syntax.

|#

; SECTION: Some Terminology Preliminary to Translation

; We now define the concepts used to describe the process of
; translation from an s-expression to a formal term.
(DEFN CORRESPONDING-NUMBERP (N)
  (IF (ZEROP N)
      (LIST1 'ZERO)
      (LIST2 'ADD1 (CORRESPONDING-NUMBERP (SUB1 N)))))
(DEFN CORRESPONDING-NEGATIVEP (N)
  (LIST2 'MINUS (CORRESPONDING-NUMBERP (NEGATIVE-GUTS N)))))
(DEFN A-D-SEQUENCEP (LST)
  (IF (NLISTP LST)
      T
      (IF (OR (EQUAL (CAR LST) (ASCII 'UPPER-A))
              (EQUAL (CAR LST) (ASCII 'UPPER-D)))
          (A-D-SEQUENCEP (CDR LST))
          F)))
(DEFN CAR-CDR-SYMBOLP (X)
  (AND (LITATOM X)
        (AND (EQUAL (CAR (UNPACK X)) (ASCII 'UPPER-C))
              (AND (EQUAL (CAR (LAST (UNPACK X))) (ASCII 'UPPER-R))
                    (A-D-SEQUENCEP (ALL-BUT-LAST (CDR (UNPACK X))))))))))

; While the text defines the A/D sequence of a CAR/CDR symbol to be
; a sequence of A's and D's, we define it to be a sequence of the
; ASCII codes of A and of D.
(DEFN A-D-SEQUENCE (X)
  (ALL-BUT-LAST (CDR (UNPACK X))))
(DEFN CAR-CDR-NEST (LST X)
  (IF (NLISTP LST)
      X
      (IF (EQUAL (CAR LST) (ASCII 'UPPER-A))
          (LIST2 'CAR (CAR-CDR-NEST (CDR LST) X))
          (LIST2 'CDR (CAR-CDR-NEST (CDR LST) X)))))

; Ultimately we will define the translation process. This will
; involve us in consing together the translations of various com-
; ponents of an s-expression, after making sure those components
; are well-formed. We signal ill-formed input by returning F in-
; stead of (the quotation of) the formal term. To make the neces-
; sary well-formedness checks less distracting we define the fol-
; lowing function which we use to create the formal terms. It is
; like CONS, but observes the convention that if either argument is
; F the result is F.

```

```

(DEFN FCONS (X Y)
  (IF (AND X Y)
    (CONS X Y)
    F))

; We also provide ourselves with several LIST-like functions that
; respect this convention. 'Tis a pity we do not have macros in
; the Nqthm logic because we really just want to define a ``func-
; tion'' symbol that takes an arbitrary number of arguments.
(DEFN FLIST1 (X) (FCONS X NIL))
(DEFN FLIST2 (X Y) (FCONS X (FLIST1 Y)))
(DEFN FLIST3 (X Y Z) (FCONS X (FLIST2 Y Z)))
(DEFN FLIST4 (X Y Z W) (FCONS X (FLIST3 Y Z W)))
(DEFN FLIST5 (X Y Z W V) (FCONS X (FLIST4 Y Z W V)))
(DEFN FLIST6 (X Y Z W V U) (FCONS X (FLIST5 Y Z W V U)))
(DEFN FLIST7 (X Y Z W V U R) (FCONS X (FLIST6 Y Z W V U R)))

; Here is the formalization of the ``fn nest around x for lst'',
; which is presented in the text before we get to the extended
; syntax. Note that we use our FCONS convention so that if X or
; any element of LST is F, the result is F.
(DEFN FN-NEST (FN LST X)
  (IF (NLISTP LST)
    X
    (FLIST3 FN (CAR LST) (FN-NEST FN (CDR LST) X))))
(DEFN CORRESPONDING-NUMBERPS (LST)
  (IF (NLISTP LST)
    NIL
    (CONS (CORRESPONDING-NUMBERP (CAR LST))
      (CORRESPONDING-NUMBERPS (CDR LST)))))
(DEFN EXPLOSION (LST)
  (FN-NEST 'CONS
    (CORRESPONDING-NUMBERPS LST)
    (LIST1 'ZERO)))
(DEFN CORRESPONDING-LITATOM (X)
  (LIST2 'PACK
    (EXPLOSION (UNPACK X))))

; The following function tacks the *1* prefix onto the front of
; LITATOMS. Thus, (star-one-star 'true) is the LITATOM '*1*TRUE --
; EXCEPT we can't write that literal atom that way because it is
; not a symbolp. It must be written as (PACK '(42 49 42 84 82 85
; 69 . 0)) or, equivalently, '(*1*QUOTE PACK (42 49 42 84 82 85 69
; . 0)).
(DEFN STAR-ONE-STAR (X)
  (PACK (APPEND (ASCII '(ASTERISK DIGIT-ONE ASTERISK))
    (UNPACK X))))

```

```

; We need to define what it is to be a symbol. The text does this
; quite early, while introducing formal terms. Here is the
; formalization.
(DEFN ASCII-UPPER-ALPHABETICS NIL
  (ASCII '(UPPER-A UPPER-B UPPER-C UPPER-D UPPER-E UPPER-F
            UPPER-G UPPER-H UPPER-I UPPER-J UPPER-K UPPER-L
            UPPER-M UPPER-N UPPER-O UPPER-P UPPER-Q UPPER-R
            UPPER-S UPPER-T UPPER-U UPPER-V UPPER-W UPPER-X
            UPPER-Y UPPER-Z)))
(DEFN ASCII-DIGITS-AND-SIGNS NIL
  (ASCII '(DIGIT-ONE DIGIT-TWO DIGIT-THREE DIGIT-FOUR DIGIT-FIVE
            DIGIT-SIX DIGIT-SEVEN DIGIT-EIGHT DIGIT-NINE
            DOLLAR-SIGN UPARROW AMPERSAND ASTERISK UNDERSCORE
            MINUS-SIGN PLUS-SIGN EQUAL-SIGN TILDE OPEN-BRACE
            CLOSE-BRACE QUESTION-MARK LESS-THAN-SIGN
            GREATER-THAN-SIGN)))
(DEFN ALL-UPPER-ALPHABETICS-DIGITS-OR-SIGNS (L)
  (IF (NLISTP L)
      T
      (AND (OR (MEMBER (CAR L) (ASCII-UPPER-ALPHABETICS))
                (MEMBER (CAR L) (ASCII-DIGITS-AND-SIGNS)))
            (ALL-UPPER-ALPHABETICS-DIGITS-OR-SIGNS (CDR L)))))
(DEFN LEGAL-CHAR-CODE-SEQ (L)
  (AND (LISTP L)
        (AND (EQUAL (CDR (LAST L)) 0)
              (AND (MEMBER (CAR L) (ASCII-UPPER-ALPHABETICS))
                    (ALL-UPPER-ALPHABETICS-DIGITS-OR-SIGNS (CDR L))))))
(DEFN SYMBOLP (X)
  (AND (LITATOM X)
        (LEGAL-CHAR-CODE-SEQ (UNPACK X))))

```

```

; SECTION: Formalizing Aspects of the History

```

```

; At this point in the text we define well-formedness and trans-
; lation. But the text delays until later the definitions of
; several concepts used. These delayed concepts are: QUOTE
; notation, the *1*QUOTE escape mechanism, and abbreviated FORs.
; We have to formalize these before we can define translation.
; Unfortunately, before we can formalize *1*QUOTE notation we must
; formalize certain concepts relating to the ``history'' in which
; the translation is occurring. For example, (FN X Y) is well-
; formed and has a translation only if FN is a function symbol of
; arity 2. We must formalize arity. We must also formalize the
; notions of what it is to be a shell constructor, a base object,
; and to satisfy the ``type- restrictions'' of a shell.

```

```

; But functions that depend upon the ``current history'' cannot be
; defined unless we wish to make the ``current history'' an ex-
; plicit object in the formalization. That is a reasonable thing
; to do but is beyond the scope of this book because the only DEFN
; events (for example) permitted in a history are those that are
; admissible, which involves the notion of proof. Thus, to formal-

```

```
; ize histories accurately we would have to formalize the rules of
; inference (not just the syntax) and what it is to be a theorem.
; This has been done for other logics in the Nqthm logic. See for
; example Shankar's work in /examples/shankar or the bibliographic
; entries for Shankar.
```

```
; Rather than formalize histories in order to formalize the syntax,
; we will formalize the syntax for a particular history. The his-
; tory we choose is the GROUND-ZERO history extended with one il-
; lustrative user-defined shell, the stacks as illustrated in the
; text, plus one illustrative user-defined function, CONCAT, which
; is just APPEND by another name. We introduce these functions be-
; low into the theory. They are nowhere used in this development
; and thus by searching for references to them you can see every-
; thing that must be known about a function symbol for it to be
; usable in the extended syntax.
```

```
; An irrelevant shell added to illustrate user-defined shells:
```

```
(ADD-SHELL PUSH EMPTY-STACK STACKP
  ((TOP (ONE-OF NUMBERP) ZERO)
   (POP (ONE-OF STACKP) EMPTY-STACK)))
```

```
; An irrelevant function added to illustrate user-defined
; functions:
```

```
(DEFN CONCAT (X Y)
  (IF (NLISTP X) Y (CONS (CAR X) (CONCAT (CDR X) Y))))
```

```
; So consider the GROUND-ZERO logic extended with the two logical
; acts above. We will now define some functions that answer ques-
; tions about that history, e.g., what are the base function
; symbols?
```

```
; The following function returns non-F if its argument is (the quo-
; tation of) a shell base function symbol. When it returns non-F
; it actually returns (the quotation of) the recognizer for the
; relevant shell.
```

```
(DEFN SHELL-BASE-FUNCTION (X)
  (IF (EQUAL X 'TRUE) 'TRUEP
      (IF (EQUAL X 'FALSE) 'FALSEP
          (IF (EQUAL X 'ZERO) 'NUMBERP
              (IF (EQUAL X 'EMPTY-STACK) 'STACKP
                  F)))))
```

```
; The next function is the analogous one for shell constructor
; function symbols.
```

```
(DEFN SHELL-CONSTRUCTOR-FUNCTION (X)
  (IF (EQUAL X 'ADD1) 'NUMBERP
      (IF (EQUAL X 'CONS) 'LISTP
          (IF (EQUAL X 'PACK) 'LITATOM
              (IF (EQUAL X 'MINUS) 'NEGATIVEP
                  (IF (EQUAL X 'PUSH) 'STACKP
                      F)))))
```



```
; The following function returns the list of type restrictions for
; a shell constructor. The list is in 1:1 correspondence with the
; accessor names, i.e., the arguments of the constructor. For ex-
; ample, for PUSH the result is '((ONE-OF NUMBERP) (ONE-OF STACKP))
; which tells us the first argument to PUSH must be a NUMBERP and
; the second must be a STACKP.
```

```
(DEFN SHELL-TYPE-RESTRICTIONS (X)
  (IF (OR (EQUAL X 'ADD1)
          (EQUAL X 'MINUS))
      (LIST1 (LIST2 'ONE-OF 'NUMBERP))
      (IF (EQUAL X 'CONS)
          (LIST2 (LIST1 'NONE-OF) (LIST1 'NONE-OF))
          (IF (EQUAL X 'PACK)
              (LIST1 (LIST1 'NONE-OF))
              (IF (EQUAL X 'PUSH)
                  (LIST2 (LIST2 'ONE-OF 'NUMBERP)
                        (LIST2 'ONE-OF 'STACKP))
                  F))))))
```

```
; This function takes a shell constructor or base function symbol
; and determines whether it satisfies a given type-restriction.
; Thus, FN might be ADD1 and TYPE-RESTRICTION might be '(ONE-OF
; NUMBERP) -- in which case SATISFIES returns T. If the type
; restriction were '(ONE-OF STACKP), SATISFIES would return F for
; FN 'ADD1.
```

```
(DEFN SATISFIES (FN TYPE-RESTRICTION)
  (IF (EQUAL (CAR TYPE-RESTRICTION) 'ONE-OF)
      (MEMBER (IF (SHELL-BASE-FUNCTION FN)
                  (SHELL-BASE-FUNCTION FN)
                  (SHELL-CONSTRUCTOR-FUNCTION FN))
              (CDR TYPE-RESTRICTION))
      (NOT (MEMBER (IF (SHELL-BASE-FUNCTION FN)
                      (SHELL-BASE-FUNCTION FN)
                      (SHELL-CONSTRUCTOR-FUNCTION FN))
                  (CDR TYPE-RESTRICTION)))))
```

```
; This function takes a list of function symbols and a list of type
; restrictions in 1:1 correspondence and determines whether all of
; the type restrictions are satisfied by the corresponding function
; symbols.
```

```
(DEFN ALL-SATISFY (FN-LST TYPE-RESTRICTION-LST)
  (IF (NLISTP FN-LST)
      T
      (AND (SATISFIES (CAR FN-LST) (CAR TYPE-RESTRICTION-LST))
            (ALL-SATISFY (CDR FN-LST) (CDR TYPE-RESTRICTION-LST)))))
```

```
; The next two functions define the arity of each of the function
; symbols in the particular history we are considering. We use the
; extended syntax here, namely QUOTE notation, to write down the
; alist. Readers using this formalization as a means of mastering
; the notation should simply understand that as a result of the
; definition below, (ARITY 'IF) = 3, (ARITY 'STACKP) = 1 and (ARITY
; 'ANY-NEW-SYMBOL) = F. That is, if (fn . n) appears in the defi-
```

```

; nition of ARITY-ALIST then (ARITY 'fn) = n. Otherwise, (ARITY
; 'fn) = F.
(DEFN ARITY-ALIST NIL
  '((IF . 3) (EQUAL . 2) (COUNT . 1) (FALSE . 0)
    (FALSEP . 1) (TRUE . 0) (TRUEP . 1) (NOT . 1)
    (AND . 2) (OR . 2) (IMPLIES . 2) (ADD1 . 1)
    (NUMBERP . 1) (SUB1 . 1) (ZERO . 0) (LESSP . 2)
    (GREATERP . 2) (LEQ . 2) (GEQ . 2) (ZEROP . 1)
    (FIX . 1) (PLUS . 2) (PACK . 1) (LITATOM . 1)
    (UNPACK . 1) (CONS . 2) (LISTP . 1) (CAR . 1)
    (CDR . 1) (NLISTP . 1) (MINUS . 1) (NEGATIVEP . 1)
    (NEGATIVE-GUTS . 1) (DIFFERENCE . 2) (TIMES . 2)
    (QUOTIENT . 2) (REMAINDER . 2) (MEMBER . 2)
    (IFF . 2) (ORD-LESSP . 2) (ORDINALP . 1)
    (ASSOC . 2) (PAIRLIST . 2) (SUBRP . 1)
    (APPLY-SUBR . 2) (FORMALS . 1) (BODY . 1)
    (FIX-COST . 2) (STRIP-CARS . 1) (SUM-CDRS . 1)
    (V&C$ . 3) (V&C-APPLY$ . 2) (APPLY$ . 2)
    (EVAL$ . 3) (QUANTIFIER-INITIAL-VALUE . 1)
    (ADD-TO-SET . 2) (APPEND . 2) (MAX . 2)
    (UNION . 2) (QUANTIFIER-OPERATION . 3) (FOR . 6)
    (PUSH . 2) (EMPTY-STACK . 0) (STACKP . 1) (TOP . 1)
    (POP . 2) (CONCAT . 2)))
(DEFN ARITY (FN)
  (IF (ASSOC FN (ARITY-ALIST))
    (CDR (ASSOC FN (ARITY-ALIST)))
    F))

; That concludes the definitions of history dependent concepts. We
; now return to the text.

; SECTION: QUOTE Notation and the *1*QUOTE Escape

; The essence of the QUOTE notation is the notion of ``explicit
; value descriptor.'' In the text, that notion is defined in terms
; of ``explicit value escape descriptor'' without the latter notion
; being defined. We then define the latter notion and its defi-
; nition involves the former. Thus, the two are mutually recur-
; sive. We formalize the mutual recursion with our traditional FLG
; argument. Normally, one would expect one value of the flag to
; indicate we were defining ``explicit value descriptor'' and the
; other value to indicate ``explicit value escape descriptor.''
; But it turns out that the basic form of mutual recursion here is
; ``explicit value descriptor'' versus ``list of explicit value
; descriptors'' and the notion of ``explicit value escape de-
; scriptor'' is written ``in-line.''

; So when FLG is T below, we check that X is an explicit value
; descriptor and if so we return the formal term it denotes. We
; return F if X is not such a descriptor. When FLG is F we check
; that X is a list of explicit value descriptors and we return
; either the list of denoted formal terms or F if X fails to be a
; list of descriptors.

```

```
; The following three events have no logical significance.  They
; do, however, permit Nqthm to process the next definition without
; inordinate delay.  We disable all function definitions except
; CADRN and CDRN.
```

```
(SET-STATUS PRE-EXPLICIT-VALUE-DESCRIPTOR
  T
  ((DEFN DISABLE) (OTHERWISE AS-IS)))
```

```
(ENABLE CADRN)
(ENABLE CDRN)
(DEFN EXPLICIT-VALUE-DESCRIPTOR (FLG X)
  (IF FLG
```

```
; Here we define what it is for X to be an explicit value de-
; scriptor and what formal term it denotes.
```

```
  (IF (NLISTP X)
    (IF (INTEGERP X)
      (IF (NUMBERP X)
        (CORRESPONDING-NUMBERP X)
        (CORRESPONDING-NEGATIVEP X))
      (IF (EQUAL X (STAR-ONE-STAR 'TRUE))
        (LIST1 'TRUE)
        (IF (EQUAL X (STAR-ONE-STAR 'FALSE))
          (LIST1 'FALSE)
          (IF (SYMBOLP X)
            (CORRESPONDING-LITATOM X)
            F))))
    (IF (EQUAL (CAR X) (STAR-ONE-STAR 'QUOTE))
```

```
; The test of the following IF contains the formalization of
; ``explicit value escape descriptor.``
```

```
  (IF (AND (OR (SHELL-CONSTRUCTOR-FUNCTION (CADRN 1 X))
    (SHELL-BASE-FUNCTION (CADRN 1 X)))
    (AND (EQLEN (CDRN 2 X) (ARITY (CADRN 1 X)))
    (AND (EQUAL (CDR (LAST X)) NIL)
    (AND (NOT (EQUAL (CADRN 1 X) 'ADD1))
    (AND (NOT (EQUAL (CADRN 1 X) 'ZERO))
    (AND (NOT (EQUAL (CADRN 1 X) 'CONS))
    (AND (EXPLICIT-VALUE-DESCRIPTOR F (CDRN 2 X))
    (AND
    (IF (SHELL-CONSTRUCTOR-FUNCTION (CADRN 1 X))
    (ALL-SATISFY
    (STRIP-CARS
    (EXPLICIT-VALUE-DESCRIPTOR F
    (CDRN 2 X)))
    (SHELL-TYPE-RESTRICTIONS (CADRN 1 X)))
    T)
    (IF (EQUAL (CADRN 1 X) 'PACK)
    (NOT (LEGAL-CHAR-CODE-SEQ (CADRN 2 X)))
    (IF (EQUAL (CADRN 1 X) 'MINUS)
```

```

                                (EQUAL (CADRN 2 X) (ZERO))
                                T)))))))))
      (CONS (CADRN 1 X)
            (EXPLICIT-VALUE-DESCRIPTOR F (CDRN 2 X)))
      F)
  (IF (DOTTED-PAIR X)
      (FLIST3 'CONS
              (EXPLICIT-VALUE-DESCRIPTOR T (CAR X))
              (EXPLICIT-VALUE-DESCRIPTOR T (CADRN 2 X)))
      (IF (SINGLETON X)
          (FLIST3 'CONS
                  (EXPLICIT-VALUE-DESCRIPTOR T (CAR X))
                  (CORRESPONDING-LITATOM NIL))
          (FLIST3 'CONS
                  (EXPLICIT-VALUE-DESCRIPTOR T (CAR X))
                  (EXPLICIT-VALUE-DESCRIPTOR T (CDR X))))))

; Here we define what it is for X to be a list of explicit value
; descriptors and the list of terms denoted by them.

  (IF (NLISTP X)
      NIL
      (FCONS (EXPLICIT-VALUE-DESCRIPTOR T (CAR X))
              (EXPLICIT-VALUE-DESCRIPTOR F (CDR X)))))

(SET-STATUS POST-EXPLICIT-VALUE-DESCRIPTOR
  T
  ((DEFN ENABLE) (OTHERWISE AS-IS)))

; The function QT is just a convenient way to refer to the explicit
; value term denoted by an explicit value descriptor (or F if its
; argument is not such a descriptor). Thus, (QT 'ABC) is '(PACK
; (CONS 65 (CONS 66 (CONS 67 0))))), except that the integers are
; actually ADD1 nests. Read ``quotation'' for QT.
(DEFN QT (X) (EXPLICIT-VALUE-DESCRIPTOR T X))

; SECTION: In Support of COND, CASE, and LET

; The next batch of functions are all involved in the translation
; of COND, CASE, and LET. We are interested in recognizing lists
; of doublets, e.g., ((w1 v1) (w2 v2) ...), the absence of
; duplication among the wi, etc.
(DEFN DOUBLET (LST)
  (IF (NLISTP LST)
      (EQUAL LST NIL)
      (AND (EQLEN (CAR LST) 2)
            (DOUBLET (CDR LST)))))
(DEFN DUPLICATESP (LST)
  (IF (NLISTP LST)
      F
      (IF (MEMBER (CAR LST) (CDR LST))
          T
          (DUPLICATESP (CDR LST)))))

```

```

(DEFN STRIP-CADRS (LST)
  (IF (NLISTP LST)
      NIL
      (CONS (CADRN 1 (CAR LST))
            (STRIP-CADRS (CDR LST)))))
(DEFN SYMBOLPS (LST)
  (IF (NLISTP LST)
      T
      (AND (SYMBOLP (CAR LST))
            (SYMBOLPS (CDR LST)))))

; This function applies the substitution ALIST to TERM (FLG=T) or
; to a list of terms (FLG=F).
(DEFN SUBLIS-VAR (FLG ALIST TERM)
  (IF FLG
      (IF (NLISTP TERM)
          (IF (ASSOC TERM ALIST)
              (CDR (ASSOC TERM ALIST))
              TERM)
          (IF (EQUAL (CAR TERM) 'QUOTE)
              TERM
              (CONS (CAR TERM)
                    (SUBLIS-VAR F ALIST (CDR TERM)))))
      (IF (NLISTP TERM)
          NIL
          (CONS (SUBLIS-VAR T ALIST (CAR TERM))
                (SUBLIS-VAR F ALIST (CDR TERM)))))
  ((LESSP (COUNT TERM)))))

; SECTION: In Support of FOR

; The text delays the discussion of FOR statements until after V&C$
; has been presented. We have to deal with them now. The fol-
; lowing functions access or check certain parts of an abbreviated
; FOR.
(DEFN ABBREVIATED-FOR-VAR (X) (CADRN 1 X))
(DEFN ABBREVIATED-FOR-RANGE (X) (CADRN 3 X))
(DEFN ABBREVIATED-FOR-WHEN (X)
  (IF (EQLEN X 8)
      (CADRN 5 X)
      'T))
(DEFN ABBREVIATED-FOR-OP (X)
  (IF (EQLEN X 8)
      (CADRN 6 X)
      (CADRN 4 X)))
(DEFN ABBREVIATED-FOR-BODY (X) (CAR (LAST X)))
(DEFN FOR-OPERATIONP (X)
  (OR (EQUAL X 'ADD-TO-SET)
      (OR (EQUAL X 'ALWAYS)
          (OR (EQUAL X 'APPEND)
              (OR (EQUAL X 'COLLECT)
                  (OR (EQUAL X 'COUNT)
                      'T))))))

```

```

      (OR (EQUAL X 'DO-RETURN)
          (OR (EQUAL X 'EXISTS)
              (OR (EQUAL X 'MAX)
                  (OR (EQUAL X 'SUM)
                      (OR (EQUAL X 'MULTIPLY)
                          (EQUAL X 'UNION))))))))))

; We now define the function that recognizes an abbreviated FOR.
(DEFN ABBREVIATED-FORP (X)
  (AND (LISTP X)
        (AND (EQUAL (CAR X) 'FOR)
              (AND (OR (EQLEN X 8)
                        (EQLEN X 6))
                    (AND (SYMBOLP (ABBREVIATED-FOR-VAR X))
                        (AND (NOT (EQUAL (ABBREVIATED-FOR-VAR X) NIL))
                            (AND (NOT (EQUAL (ABBREVIATED-FOR-VAR X) 'T))
                                (AND (NOT (EQUAL (ABBREVIATED-FOR-VAR X) 'F))
                                    (AND (EQUAL (CADRN 2 X) 'IN)
                                        (AND (OR (EQLEN X 6)
                                                (EQUAL (CADRN 4 X) 'WHEN))
                                            (FOR-OPERATIONP
                                                (ABBREVIATED-FOR-OP X))))))))))))))

; To translate an abbreviated FOR we must sort the list of vari-
; ables used in the WHEN clause and the BODY.
(DEFN ALPHABETIC-LESSP1 (L1 L2)
  (IF (NLISTP L1)
      T
      (IF (NLISTP L2)
          F
          (IF (LESSP (CAR L1) (CAR L2))
              T
              (IF (EQUAL (CAR L1) (CAR L2))
                  (ALPHABETIC-LESSP1 (CDR L1) (CDR L2))
                  F))))))
(DEFN ALPHABETIC-LESSP (X Y)
  (ALPHABETIC-LESSP1 (UNPACK X)
                     (UNPACK Y)))
(DEFN ALPHABETIC-INSERT (X L)
  (IF (NLISTP L)
      (LIST1 X)
      (IF (ALPHABETIC-LESSP X (CAR L))
          (CONS X L)
          (CONS (CAR L)
                 (ALPHABETIC-INSERT X (CDR L))))))
(DEFN ALPHABETIZE (L)
  (IF (NLISTP L)
      L
      (ALPHABETIC-INSERT (CAR L)
                          (ALPHABETIZE (CDR L)))))

; To collect the variable symbols that occur in a term (or list of
; terms) we use ALL-VARS.

```

```

(DEFN ALL-VARS (FLG X)
  (IF FLG
    (IF (NLISTP X)
      (CONS X NIL)
      (ALL-VARS F (CDR X)))
    (IF (NLISTP X)
      NIL
      (UNION (ALL-VARS T (CAR X))
              (ALL-VARS F (CDR X))))))
(DEFN STANDARD-ALIST (VARS)
  (IF (NLISTP VARS)
    (QT NIL)
    (LIST3 'CONS
           (LIST3 'CONS
                  (QT (CAR VARS))
                  (CAR VARS))
           (STANDARD-ALIST (CDR VARS)))))
(DEFN DELETE (X L)
  (IF (NLISTP L)
    L
    (IF (EQUAL X (CAR L))
      (CDR L)
      (CONS (CAR L) (DELETE X (CDR L))))))
(DEFN MAKE-ALIST (VAR WHEN BODY)
  (STANDARD-ALIST
   (ALPHABETIZE
    (DELETE VAR
             (UNION (ALL-VARS T WHEN)
                     (ALL-VARS T BODY))))))

; The following lemmas are used in the justification of the defi-
; nition of TRANSLATE.

(PROVE-LEMMA LESSP-ABBREVIATED-FOR-RANGE (REWRITE)
  (IMPLIES (EQUAL (CAR X) 'FOR)
            (LESSP (COUNT (ABBREVIATED-FOR-RANGE X)) (COUNT X))))

(PROVE-LEMMA LESSP-ABBREVIATED-FOR-WHEN (REWRITE)
  (IMPLIES (EQUAL (CAR X) 'FOR)
            (LESSP (COUNT (ABBREVIATED-FOR-WHEN X)) (COUNT X))))

(PROVE-LEMMA LESSP-LAST (REWRITE)
  (NOT (LESSP (COUNT X) (COUNT (LAST X)))))

(PROVE-LEMMA LESSP-ABBREVIATED-FOR-BODY (REWRITE)
  (IMPLIES (EQUAL (CAR X) 'FOR)
            (LESSP (COUNT (ABBREVIATED-FOR-BODY X)) (COUNT X))))

(PROVE-LEMMA LESSP-COUNT-STRIP-CARS (REWRITE)
  (IMPLIES (DOUBLET'S LST)
            (NOT (LESSP (COUNT LST)
                        (COUNT (STRIP-CARS LST))))))

```

```

(PROVE-LEMMA LESSP-COUNT-STRIP-CADRS (REWRITE)
  (IMPLIES (DOUBLET LST)
    (NOT (LESSP (COUNT LST)
      (COUNT (STRIP-CADRS LST))))))

(PROVE-LEMMA LISTP-CDDR-X-COUNT-X (REWRITE)
  (IMPLIES (LISTP (CDDR X))
    (EQUAL (COUNT X)
      (ADD1
        (ADD1
          (PLUS (COUNT (CAR X))
            (PLUS (COUNT (CADR X))
              (COUNT (CDDR X))))))))))

(PROVE-LEMMA LISTP-CDDDR-X-COUNT-X (REWRITE)
  (IMPLIES (LISTP (CDDDR X))
    (EQUAL (COUNT X)
      (ADD1
        (ADD1
          (ADD1
            (PLUS (COUNT (CAR X))
              (PLUS (COUNT (CADR X))
                (PLUS (COUNT (CADDR X))
                  (COUNT (CDDDR X))))))))))))))

; SECTION: Translation

; We disable all definitions except CADR and CDR.

(SET-STATUS PRE-TRANSLATE T ((DEFN DISABLE)(OTHERWISE AS-IS)))
(ENABLE CADR)
(ENABLE CDR)

; Here, finally, is the formalization of what it is to be well-
; formed and what the translation of a well-formed term is. If
; TRANSLATE (FLG=T) returns F, then X is not well-formed; other-
; wise, TRANSLATE (FLG=T) returns the (quotation of the) formal
; term denoted by X. Because a formal term is either a variable
; symbol (i.e., LITATOM) or function application (i.e., LISTP), an
; answer of F unambiguously identifies the input as ill-formed.
; When FLG=F, TRANSLATE operates on a list of purported terms and
; returns either F, meaning at least one of the elements is
; ill-formed, or returns the list of their translations.
(DEFN TRANSLATE (FLG X)
  (IF FLG

; When FLG = T, we are translating a single term, X.

    (IF (NLISTP X)
      (IF (INTEGERP X)
        (QT X)
        (IF (SYMBOLP X)

```



```

      (IF (EQUAL X 'T) (LIST1 'TRUE)
        (IF (EQUAL X 'F) (LIST1 'FALSE)
          (IF (EQUAL X NIL) (QT NIL)
            X)))
    F))
(IF (DOTTED-S-EXPRESSION X)
  F
  (IF (EQUAL (CAR X) 'QUOTE)
    (IF (EQLEN X 2)
      (QT (CADRN 1 X))
      F)
    (IF (EQUAL (CAR X) 'COND)
      (IF (AND (EQLEN X 2)
        (AND (EQLEN (CADRN 1 X) 2)
          (EQUAL (CAR (CADRN 1 X)) 'T)))
        (TRANSLATE T (CADRN 1 (CADRN 1 X)))
        (IF (AND (EQLEN (CADRN 1 X) 2)
          (AND (NOT (EQUAL (CAR (CADRN 1 X)) 'T))
            (LISTP (CDRN 2 X))))
          (FLIST4 'IF
            (TRANSLATE T (CAR (CADRN 1 X)))
            (TRANSLATE T (CADRN 1 (CADRN 1 X)))
            (TRANSLATE T (CONS 'COND (CDRN 2 X))))
          F))
      (IF (EQUAL (CAR X) 'CASE)
        (IF (AND (EQLEN X 3)
          (AND (EQLEN (CADRN 2 X) 2)
            (AND (EQUAL (CAR (CADRN 2 X)) 'OTHERWISE)
              (TRANSLATE T (CADRN 1 X))))))
          (TRANSLATE T (CADRN 1 (CADRN 2 X)))
          (IF (AND (EQLEN (CADRN 2 X) 2)
            (AND (LISTP (CDRN 3 X))
              (NOT (MEMBER (CAR (CADRN 2 X))
                (STRIP-CARS (CDRN 3 X))))))
            (FLIST4 'IF
              (FLIST3 'EQUAL
                (TRANSLATE T (CADRN 1 X))
                (QT (CAR (CADRN 2 X)))
                (TRANSLATE T (CADRN 1 (CADRN 2 X)))
                (TRANSLATE T (CONS 'CASE
                  (CONS (CADRN 1 X)
                    (CDRN 3 X))))))
              F))
          (IF (EQUAL (CAR X) 'LET)

```

; The first test below, on DOUBLET, belongs inside the AND nest.
; But if it is there it is not among the governing terms of the
; recursions in that nest and the termination argument breaks down.

```

(IF (DOUBLET (CADRN 1 X))
  (IF (AND (EQLEN X 3)
    (AND (TRANSLATE F (STRIP-CARS (CADRN 1 X)))
      (AND (TRANSLATE F (STRIP-CARS (CADRN 1 X)))

```

```

(AND (TRANSLATE T (CADRN 2 X))
      (AND
        (SYMBOLPS
          (TRANSLATE F (STRIP-CARS (CADRN 1 X))))
        (NOT
          (DUPLICATESP
            (TRANSLATE
              F (STRIP-CARS (CADRN 1 X)))))))))
(SUBLIS-VAR
  T
  (PAIRLIST
    (TRANSLATE F (STRIP-CARS (CADRN 1 X)))
    (TRANSLATE F (STRIP-CADRS (CADRN 1 X))))
  (TRANSLATE T (CADRN 2 X)))
F)
F)
(IF (NOT (TRANSLATE F (CDR X)))
  F
  (IF (OR (EQUAL (CAR X) NIL)
    (OR (EQUAL (CAR X) 'T)
      (EQUAL (CAR X) 'F)))
    F
    (IF (EQUAL (CAR X) 'LIST)
      (FN-NEST 'CONS
        (TRANSLATE F (CDR X))
        (QT NIL))
      (IF (EQUAL (CAR X) 'LIST*)
        (IF (EQLEN X 1)
          F
          (FN-NEST 'CONS
            (ALL-BUT-LAST (TRANSLATE F (CDR X)))
            (CAR (LAST (TRANSLATE F (CDR X))))))
        (IF (CAR-CDR-SYMBOLP (CAR X))
          (IF (EQLEN X 2)
            (CAR-CDR-NEST (A-D-SEQUENCE (CAR X))
              (TRANSLATE T (CADRN 1 X)))
            F)
          (IF (EQLEN (CDR X) (ARITY (CAR X)))
            (FCONS (CAR X) (TRANSLATE F (CDR X)))
            (IF (EQUAL (CAR X) 'FOR)
              (IF (ABBREVIATED-FORP X)
                (FLIST7 'FOR
                  (QT (ABBREVIATED-FOR-VAR X))
                  (TRANSLATE T (ABBREVIATED-FOR-RANGE X))
                  (QT (TRANSLATE T
                    (ABBREVIATED-FOR-WHEN X)))
                  (QT (ABBREVIATED-FOR-OP X))
                  (QT (TRANSLATE T
                    (ABBREVIATED-FOR-BODY X)))
                  (MAKE-ALIST
                    (ABBREVIATED-FOR-VAR X)
                    (TRANSLATE T
                      (ABBREVIATED-FOR-WHEN X)))
                )
              )
            )
          )
        )
      )
    )
  )

```

```

      (TRANSLATE T
        (ABBREVIATED-FOR-BODY X))))
    F)
  (IF (AND (LESSP 2 (LENGTH (CDR X)))
    (OR (EQUAL (CAR X) 'AND)
      (OR (EQUAL (CAR X) 'OR)
        (OR (EQUAL (CAR X) 'PLUS)
          (EQUAL (CAR X) 'TIMES)))))
    (FN-NEST (CAR X)
      (ALL-BUT-LAST (TRANSLATE F (CDR X)))
      (CAR (LAST (TRANSLATE F (CDR X)))))
    F)))))))))

; This is the case where FLG = F and we are translating a list of
; terms, X.

  (IF (NLISTP X)
    NIL
    (FCONS (TRANSLATE T (CAR X))
      (TRANSLATE F (CDR X)))))
  ((LESSP (COUNT X))))

(SET-STATUS POST-TRANSLATE T ((DEFN ENABLE)(OTHERWISE AS-IS)))

; SECTION: The Extended Syntax

; Finally, here is EXSYN, which reads an s-expression from a stream
; of ASCII character codes and translates it into a formal term or
; returns F if the stream is not the display of a term in the ex-
; tended syntax.
(DEFN EXSYN (STREAM)
  (IF (READABLE (READ-TOKEN-TREE STREAM) 0)
    (TRANSLATE T (READMACRO-EXPANSION (READ-TOKEN-TREE STREAM)))
    F))

; EXSYN returns F if the stream cannot be parsed. The explanation
; of this remark is that an unparsable stream causes READ-TOKEN-
; TREE to return F and READABLE returns F on that input.

; SECTION: Slightly Abbreviated Formal Terms

; It is exceedingly difficult to read the output of TRANSLATE and
; EXSYN because quoted literal atoms and numbers are exploded.
; Thus 'ABC translates to (PACK (CONS (ADD1 ...) ...)) where the
; ellipses are very large nests of CONSES and ADD1s. Below, we
; develop a function that can be used to massage the output of
; TRANSLATE to introduce QUOTE notation for LITATOMs and to intro-
; duce the normal decimal representation for ADD1 nests. While
; this convention is employed in Chapter 4, e.g., when we exhibit
; two token trees with the same translation, the concepts
; formalized below are not defined in the text.

```

; If X is an ADD1-nest n deep with a (ZERO) at the bottom, we
 ; return n; otherwise F. The variable I is used as an accumulator
 ; and should be 0 at the top-level.

```
(DEFN ADD1-NESTP (X I)
  (IF (NLISTP X)
      F
      (IF (AND (EQUAL (CAR X) 'ZERO)
                (EQLEN X 1))
          I
          (IF (AND (EQUAL (CAR X) 'ADD1)
                    (EQLEN X 2))
              (ADD1-NESTP (CADRN 1 X) (ADD1 I))
              F))))
  (DEFN CONS-ADD1-NESTP (X)
    (IF (NLISTP X)
        F
        (IF (AND (EQUAL (CAR X) 'ZERO)
                  (EQLEN X 1))
            0
            (IF (AND (EQUAL (CAR X) 'CONS)
                      (EQLEN X 3))
                (FCONS (ADD1-NESTP (CADRN 1 X) 0)
                        (CONS-ADD1-NESTP (CADRN 2 X)))
                F))))
    (DEFN EXPLODED-LITATOM (X)
      (IF (AND (EQLEN X 2)
                (AND (EQUAL (CAR X) 'PACK)
                      (CONS-ADD1-NESTP (CADRN 1 X))))
          (LIST2 'QUOTE (PACK (CONS-ADD1-NESTP (CADRN 1 X))))
          F))
    (DEFN ABBREV (FLG X)
      (IF FLG
          (IF (NLISTP X)
              X
              (IF (ADD1-NESTP X 0)
                  (ADD1-NESTP X 0)
                  (IF (EXPLODED-LITATOM X)
                      (EXPLODED-LITATOM X)
                      (CONS (CAR X)
                            (ABBREV F (CDR X))))))
          (IF (NLISTP X)
              NIL
              (CONS (ABBREV T (CAR X))
                    (ABBREV F (CDR X))))))
```

; Here is a version of EXSYN that uses abbreviations.

```
(DEFN AEXSYN
  (STREAM)
  (IF (READABLE (READ-TOKEN-TREE STREAM) 0)
      (ABBREV T
        (TRANSLATE T
          (READMACRO-EXPANSION (READ-TOKEN-TREE STREAM))))
      F))
```

#|

Here are Common Lisp interfaces to EXSYN and AEXSYN.

```
(defun exsyn (string)
  (*1*exsyn (ascii string)))
```

```
(defun aexsyn (string)
  (*1*aexsyn (ascii string)))
```

|#

```
; We conclude by making a compiled library containing the current
; data base. If one executes (NOTE-LIB ".../examples/basic/parser"
; T) in Nqthm, where the ellipsis is meant to be the local
; directory containing our examples subdirectory, then one can use
; R-LOOP to execute these functions. For example, one can type to
; R-LOOP:
```

```
;(AEXSYN
; (ASCII
; '(OPEN-PAREN
;   LOWER-A LOWER-D LOWER-D DIGIT-ONE
;   SPACE
;   NUMBER-SIGN VERTICAL-BAR
;   UPPER-C LOWER-O LOWER-M LOWER-M LOWER-E LOWER-N LOWER-T
;   VERTICAL-BAR NUMBER-SIGN
;   LOWER-X
;   CLOSE-PAREN)))
```

```
; and get the result '(ADD X).
```

```
; This is sufficiently cumbersome that we find the Lisp interface
; functions much more convenient. If the ``defuns'' in this file
; are executed in an acceptable Common Lisp, then it is possible to
; type to Common Lisp (rather than R-LOOP):
```

```
; (aexsyn "(add1 #|Comment|#x)")
```

```
; and get the result (ADD1 X). The Lisp routine aexsyn actually
; executes our logical function AEXSYN but it first converts the
; string argument into a list of ASCII characters.
```

```
(MAKE-LIB "parser" T)
```

Appendix II

The Primitive Shell Axioms

In this appendix we present the axioms defining the primitive shells, namely the natural numbers, the ordered pairs, the literal atoms, and the negative integers. In our treatment of the formal theory in Chapter 4, we used the shell principle to add these axioms. This appendix is provided both for those readers who wish to understand the shell principle by inspecting the axioms generated by its schemas and for those who wish to see the basic axioms of our logic. The reader who eschews use of the shell principle can eliminate it entirely from the logic by substituting the axioms shown below for the four invocations of the principle in Chapter 4.

The axioms shown below are mild simplifications of those generated by the invocations of the shell principle. For example, in the schema for axiom 11 of the Shell Principle the subterm

$$\begin{aligned} &(\text{PLUS } (\text{COUNT } (\text{ac}_1 \text{ X})) \\ &\quad \dots \\ &\quad (\text{COUNT } (\text{ac}_n \text{ X}))) \otimes (\text{ZERO}) \end{aligned}$$

appears. When instantiated to generate the axioms for ordered pairs the subterm generated is

$$\begin{aligned} &(\text{PLUS } (\text{COUNT } (\text{CAR } \text{X})) \\ &\quad (\text{PLUS } (\text{COUNT } (\text{CDR } \text{X})) \\ &\quad (\text{ZERO}))) \end{aligned}$$

However, here we display the equivalent term:

```
(PLUS (COUNT (CAR X))
      (COUNT (CDR X)))
```

Similarly, when no base object is supplied or the type restrictions are (**NONE-OF**), the axioms generated can be greatly simplified by applying such propositional transformations as $(\text{OR } P \text{ } F) \leftrightarrow P$ and $(\text{IF } F \text{ } X \text{ } Y) = Y$.

As explained on page 135, some shell axioms have been renumbered due to the deletion of redundant axioms in the first edition.

II.1. The Natural Numbers

The axioms added by

Shell Definition.

Add the shell **ADD1** of one argument
with base function **ZERO**,
recognizer function **NUMBERP**,
accessor function **SUB1**,
type restriction (**ONE-OF NUMBERP**),
default function **ZERO**.

are

Axiom 12.1.

```
(OR (EQUAL (NUMBERP X) T)
     (EQUAL (NUMBERP X) F))
```

Axiom 12.2.

```
(NUMBERP (ADD1 X1))
```

Axiom 12.3.

```
(NUMBERP (ZERO))
```

Axiom 12.4.

```
(NOT (EQUAL (ADD1 X1) (ZERO)))
```

Axiom 12.5.

```
(IMPLIES (AND (NUMBERP X)
               (NOT (EQUAL X (ZERO))))
          (EQUAL (ADD1 (SUB1 X)) X))
```

Axiom 12.6.

```
(IMPLIES (NUMBERP X1)
          (EQUAL (SUB1 (ADD1 X1)) X1))
```

Axiom 12.7.

```
(IMPLIES (OR (NOT (NUMBERP X))
              (OR (EQUAL X (ZERO))
                  (AND (NOT (NUMBERP X1))
                      (EQUAL X (ADD1 X1))))))
          (EQUAL (SUB1 X) (ZERO)))
```

Axiom 12.8.1.

```
(IMPLIES (NUMBERP X) (NOT (TRUEP X)))
```

Axiom 12.8.2.

```
(IMPLIES (NUMBERP X) (NOT (FALSEP X)))
```

Axiom 12.9.

```
(IMPLIES (NUMBERP X)
          (EQUAL (COUNT X)
                  (IF (EQUAL X (ZERO))
                      (ZERO)
                      (ADD1 (COUNT (SUB1 X)))))))
```

Axiom 12.10.1.

```
(AND (EQUAL (SUBRP 'ADD1) T)
      (EQUAL (APPLY-SUBR 'ADD1 L)
              (ADD1 (CAR L))))
```

Axiom 12.10.2.

```
(AND (EQUAL (SUBRP 'ZERO) T)
      (EQUAL (APPLY-SUBR 'ZERO L)
              (ZERO)))
```

Axiom 12.10.3.

```
(AND (EQUAL (SUBRP 'NUMBERP) T)
      (EQUAL (APPLY-SUBR 'NUMBERP L)
              (NUMBERP (CAR L))))
```

Axiom 12.10.4.

```
(AND (EQUAL (SUBRP 'SUB1) T)
      (EQUAL (APPLY-SUBR 'SUB1 L)
              (SUB1 (CAR L))))
```


II.2. The Ordered Pairs

The axioms added by

Shell Definition.

Add the shell **CONS** of two arguments with recognizer function **LISTP**, accessor functions **CAR** and **CDR**, default functions **ZERO** and **ZERO**.

are

Axiom 19.1.

```
(OR (EQUAL (LISTP X) T)
    (EQUAL (LISTP X) F))
```

Axiom 19.2.

```
(LISTP (CONS X1 X2))
```

Axiom 19.5.

```
(IMPLIES (LISTP X)
          (EQUAL (CONS (CAR X) (CDR X)) X))
```

Axiom 19.6.1.

```
(EQUAL (CAR (CONS X1 X2)) X1)
```

Axiom 19.6.2.

```
(EQUAL (CDR (CONS X1 X2)) X2)
```

Axiom 19.7.1.

```
(IMPLIES (NOT (LISTP X))
          (EQUAL (CAR X) (ZERO)))
```

Axiom 19.7.2.

```
(IMPLIES (NOT (LISTP X))
          (EQUAL (CDR X) (ZERO)))
```

Axiom 19.8.1.

```
(IMPLIES (LISTP X) (NOT (NUMBERP X)))
```

Axiom 19.8.2.

```
(IMPLIES (LISTP X) (NOT (TRUEP X)))
```

Axiom 19.8.3.

```
(IMPLIES (LISTP X) (NOT (FALSEP X)))
```

Axiom 19.9.

```
(IMPLIES (LISTP X)
  (EQUAL (COUNT X)
    (ADD1 (PLUS (COUNT (CAR X))
      (COUNT (CDR X))))))
```

Axiom 19.10.1.

```
(AND (EQUAL (SUBRP 'CONS) T)
  (EQUAL (APPLY-SUBR 'CONS L)
    (CONS (CAR L) (CDR L))))
```

Axiom 19.10.2.

```
(AND (EQUAL (SUBRP 'LISTP) T)
  (EQUAL (APPLY-SUBR 'LISTP L)
    (LISTP (CAR L))))
```

Axiom 19.10.3.

```
(AND (EQUAL (SUBRP 'CAR) T)
  (EQUAL (APPLY-SUBR 'CAR L)
    (CAR (CAR L))))
```

Axiom 19.10.4.

```
(AND (EQUAL (SUBRP 'CDR) T)
  (EQUAL (APPLY-SUBR 'CDR L)
    (CDR (CAR L))))
```

Schemas (3) and (4) generate the trivial axiom **T** since there is no base object.

II.3. The Literal Atoms

The axioms added by

Shell Definition.

Add the shell **PACK** of one argument
with recognizer function **LITATOM**,
accessor function **UNPACK**,
default function **ZERO**.

are

Axiom 20.1.

```
(OR (EQUAL (LITATOM X) T)
  (EQUAL (LITATOM X) F))
```

Axiom 20.2.

```
(LITATOM (PACK X1))
```

Axiom 20.5.

```
(IMPLIES (LITATOM X)
          (EQUAL (PACK (UNPACK X)) X))
```

Axiom 20.6.

```
(EQUAL (UNPACK (PACK X1)) X1)
```

Axiom 20.7.

```
(IMPLIES (NOT (LITATOM X))
          (EQUAL (UNPACK X) (ZERO)))
```

Axiom 20.8.1.

```
(IMPLIES (LITATOM X) (NOT (LISTP X)))
```

Axiom 20.8.2.

```
(IMPLIES (LITATOM X) (NOT (NUMBERP X)))
```

Axiom 20.8.3.

```
(IMPLIES (LITATOM X) (NOT (TRUEP X)))
```

Axiom 20.8.4.

```
(IMPLIES (LITATOM X) (NOT (FALSEP X)))
```

Axiom 20.9.

```
(IMPLIES (LITATOM X)
          (EQUAL (COUNT X)
                  (ADD1 (COUNT (UNPACK X)))))
```

Axiom 20.10.1.

```
(AND (EQUAL (SUBRP 'PACK) T)
      (EQUAL (APPLY-SUBR 'PACK L) (PACK (CAR L))))
```

Axiom 20.10.2.

```
(AND (EQUAL (SUBRP 'LITATOM) T)
      (EQUAL (APPLY-SUBR 'LITATOM L)
              (LITATOM (CAR L))))
```

Axiom 20.10.3.

```
(AND (EQUAL (SUBRP 'UNPACK) T)
      (EQUAL (APPLY-SUBR 'UNPACK L)
              (UNPACK (CAR L))))
```

Schemas (3) and (4) generate the trivial axiom **T** since there is no base object.

II.4. The Negative Integers

The axioms added by

Shell Definition.

Add the shell **MINUS** of one argument
with recognizer function **NEGATIVEP**,
accessor function **NEGATIVE-GUTS**,
type restriction (**ONE-OF** **NUMBERP**),
default function **ZERO**.

are

Axiom 21.1.

```
(OR (EQUAL (NEGATIVEP X) T)
    (EQUAL (NEGATIVEP X) F))
```

Axiom 21.2.

```
(NEGATIVEP (MINUS X1))
```

Axiom 21.5.

```
(IMPLIES (NEGATIVEP X)
          (EQUAL (MINUS (NEGATIVE-GUTS X)) X))
```

Axiom 21.6.

```
(IMPLIES (NUMBERP X1)
          (EQUAL (NEGATIVE-GUTS (MINUS X1)) X1))
```

Axiom 21.7.

```
(IMPLIES (OR (NOT (NEGATIVEP X))
              (AND (NOT (NUMBERP X1))
                   (EQUAL X (MINUS X1)))))
          (EQUAL (NEGATIVE-GUTS X) (ZERO)))
```

Axiom 21.8.1.

```
(IMPLIES (NEGATIVEP X) (NOT (LITATOM X)))
```

Axiom 21.8.2.

```
(IMPLIES (NEGATIVEP X) (NOT (LISTP X)))
```

Axiom 21.8.3.

```
(IMPLIES (NEGATIVEP X) (NOT (NUMBERP X)))
```

Axiom 21.8.4.

```
(IMPLIES (NEGATIVEP X) (NOT (TRUEP X)))
```

Axiom 21.8.5.

```
(IMPLIES (NEGATIVEP X) (NOT (FALSEP X)))
```

Axiom 21.9.

```
(IMPLIES (NEGATIVEP X)
  (EQUAL (COUNT X)
    (ADD1 (COUNT (NEGATIVE-GUTS X)))))
```

Axiom 21.10.1.

```
(AND (EQUAL (SUBRP 'MINUS) T)
  (EQUAL (APPLY-SUBR 'MINUS L)
    (MINUS (CAR L))))
```

Axiom 21.10.2.

```
(AND (EQUAL (SUBRP 'NEGATIVEP) T)
  (EQUAL (APPLY-SUBR 'NEGATIVEP L)
    (NEGATIVEP (CAR L))))
```

Axiom 21.10.3.

```
(AND (EQUAL (SUBRP 'NEGATIVE-GUTS) T)
  (EQUAL (APPLY-SUBR 'NEGATIVE-GUTS L)
    (NEGATIVE-GUTS (CAR L))))
```

Schemas (3) and (4) generate the trivial axiom **T** since there is no base object.

Appendix III

On the Difficulty of Proofs

In the Preface we note that the difficulty of the theorems proved by the theorem prover has grown substantially over the years. We recently quantified “complexity” and then compared some of the important landmarks over the years. While several of the complexity measures are completely artificial, we find that the numeric comparisons substantiate the claim that the proofs produced in the last few years are much deeper than those in *A Computational Logic*.

We defined six different measures of the complexity of a theorem.

- The number of lines of prettyprinted text in the “understandable statement” of the theorem. This is the number of lines of text you would have to read, starting at the Ground Zero logic, simply to understand what was being proved. We omit from consideration the definitions of “existential” functions—those functions used only to express existential quantification and whose definitions are irrelevant to the intended interpretation of the theorem. This notion is very subjective. Consider for example (**PERM L (SORT L)**), which says that the function **SORT** produces a permutation of its input. One might regard the theorem merely as saying “there exists a permutation of any **L**.” Reading the theorem that way absolves the reader from having to understand **SORT** and its subroutines while coming to an understanding of what the theorem says. Of course, to prove the theorem one must consider the particular definition of **SORT** used; in essence, these “existential” functions

are regarded as part of the proof rather than as part of the statement of the theorem.

- The “conceptual depth” of the understandable statement of the theorem. This is an entirely artificial concept attempting to measure how hard it is to understand a formula. It is defined as follows. The depth of a variable is 0; the depth of a term (**fn** $a_1 \dots a_n$) is the depth of the function symbol **fn** plus the depth of the deepest a_i . The depth of a primitive function symbol is 0; the depth of a defined function is one greater than the depth of its body. Thus, **INSERT** has depth 1, **SORT** has depth 2, and (**SORT** (**SORT** **X**)) has depth 4. Again, we omit from consideration the existential terms in a theorem.
- The maximum conceptual depth of any concept used in the proof. Some of our proofs involve concepts significantly deeper than those in the statement.
- The number of “supporters” in the proof. The supporters is the set of all function and lemma names involved in the dependency graph of the proof, except for those functions used in the understandable statement of the theorem. Thus, you would have to look at every supporter to understand the proof.
- The number of lines necessary to prettyprint all the supporters.
- The depth of the proof. This is the length of the longest branch in the dependency graph from the theorem to the axioms.

These six measures do not include the most obvious: the number of lines in the formal proof allegedly checked. We discuss this omission and give some estimates at the end of this appendix.

We have selected twelve different formalization and proof projects carried out during the nineteen year period 1973-1991. Each project was regarded (by us) as an impressive achievement at the time it was done. Each project was “crowned” by some particular theorem proved and we measured the complexity of each theorem according to all six criteria sketched above. The theorems measured are all included in the standard release of the theorem prover (see the files described on page 423). The names of the theorems and brief descriptions are given below.

- **ORDERED-SORT: (ORDERED (SORT L))**. This was the most interesting theorem proved by the “Edinburgh version” of our system, the theorem prover first released in 1973 and described in [22]. The theorem is found in example file 13. We refer to the theorem below simply as the “**SORT**” theorem, for brevity.

- **CORRECTNESS-OF-OPTIMIZING-COMPILER**: the correctness of a simple expression compiler for a push-down stack machine. The theorem was first proved by our theorem prover in 1976. The proof was described in [23] and used as one of the examples in *A Computational Logic* [24] in 1979. The theorem is found in example file 13. We refer to the theorem below as the “**COMPILER**” theorem.
- **TAUTOLOGY-CHECKER-IS-COMPLETE**: the completeness of a tautology checker for **IF**-expressions, similar to the Wang algorithm. The theorem was proved by the theorem prover in 1978 and used as one of the examples in [24] in 1979. The theorem is found in example file 13. We call it “**TAUT**” below.
- **PRIME-FACTORIZATION-UNIQUENESS**: two lists of primes with equal products are permutations of one another. The theorem was first proved by the theorem prover in 1978 and used as an example in [24]. The theorem is found in example file 13. We call it “**PRIME**” below.
- **UNSOLVABILITY-OF-THE-HALTING-PROBLEM**: no Pure Lisp program can solve the halting problem for Pure Lisp. The theorem was proved by the theorem prover in 1982 and published in [29]. The theorem is found in example file 19. We call it “**UNSOLV**” below.
- **CRYPT-INVERTS**: the RSA public key encryption algorithm is invertible. The proof, a formalization of that published in [117] by Rivest, Shamir, and Adleman, was carried out by the theorem prover in 1983 and described in [27]. The theorem is found in example file 15. We call it “**RSA**” below.
- **LAW-OF-QUADRATIC-RECIPROCITY**: Gauss’s law, often referred to as the “jewel of elementary number theory.” The mechanical proof was carried out under the direction of David Rusinoff in 1984. The theorem is found in example file 8. We call it “**GAUSS**” below.
- **FINALLY-CHURCH-ROSSER**: the Church-Rosser theorem (in its final form rather than in the intermediate, variable-free form). The proof was carried out under the direction of Natarajan Shankar in 1985 and is described in [128, 127]. The theorem is found in example file 63. We call it “**C-R**” below.
- **SOFT-RESET-WORKS**: the correctness of the FM8501 microprocessor. The mechanical proof was carried out under the direction of Warren Hunt in 1985 and is described in [73, 70]. The theorem is found in example file 36. We call it “**FM8501**” below.

- **INCOMPLETENESS-THEOREM**: Gödel's incompleteness theorem. The mechanical proof was carried out under the direction of Natarajan Shankar in 1986 and is described in [127, 129]. The theorem is found in example file 64. We call it "**GOEDEL**" below.
- **CHIP-SYSTEM=FM9001-INTERPRETER**: the correctness of the FM9001 microprocessor. The mechanical proof was carried out under the direction of Bishop Brock and Warren Hunt in 1991 and is described in [74]. The theorem measured below is actually the conjunction of nine results in the FM9001 correctness proof; the nine results are named **CHIP-SYSTEM=FM9001-INTERPRETER**, **RESET-WORKS**, **INITIAL-STATE-OKP**, **INSTANCE-THEOREM**, **V-ALU-CORRECT-NAT-INT-BUF**, **V-ALU-CORRECT-NAT**, **V-ALU-CORRECT-INT**, **CHIP-WELL-FORMED**, and **CHIP-WELL-FORMED-AFTER-INDEXED-NAMES-REMOVED**. We call the theorem "**FM9001**" below.
- **TOP-GOAL**: the correctness of the Piton compiler for the FM9001 (see [106]). The theorem is found in example file 31. The numbers computed for Piton below treat FM9001 as though it were a primitive function symbol. We call the theorem "**PITON**" below.

Table III.1

Number of Lines in Understandable Stmt						
. Concept Depth of Stmt						
. . Max Concept Depth in Proof						
. . . Number of Supporters						
. . . Lines of Supporters						
. . . Depth of Proof						
73	SORT	24	3	2	0	3
76	COMPILER	88	6	5	7	5
78	TAUT	96	6	4	31	9
78	PRIME	49	4	8	108	17
82	UNSOLV	164	2	7	18	6
83	RSA	43	4	8	172	24
84	GAUSS	48	3	8	348	37
85	C-R	114	5	42	133	16
85	FM8501	991	157	152	230	18
86	GOEDEL	864	48	40414	1741	58
91	FM9001	7850	120	128	1669	46
91	PITON	3933	92	102	3326	54

Table III.1 contains the data. By all measures, the complexity of the theorems and their proofs have increased dramatically over the years.

A few comments are in order. The correctness of the FM9001 microprocessor, at 7,850 lines, is by far the most verbose theorem proved to date. Of course, the statement of the theorem is broken up into components, e.g., the specification of the microprocessor's machine code, the specification of the hardware design language in which the design is described, etc. We discuss the decomposition of FM9001 at more length below. The verbosity of the Piton theorem, at almost 4,000 lines, is even more remarkable when one realizes that the definition of FM9001 is not counted here. Most of the Piton lines are devoted to the definition of the Piton assembly language. Finally, the 864 lines devoted to Gödel's theorem may surprise logicians since the textbook presentation of the theorem normally only consumes a few lines. Our line count includes the formal definition of the "object logic" proved incomplete. That logic is defined via a proof checker for it, written as a function in our logic.

Until FM8501 and Gödel's theorem were proved, all of our theorems were pretty understandable, with statements of depth at most 6. The depth of the statement of Gödel's theorem, 48, reflects how deep the associated proof checker is. The record depth, that for FM8501, at 157, is due to the nesting of hardware modules representing the composition of circuits. If the concept of "depth" is to be believed, we conclude that it is harder to understand what is said about FM8501 than it is to understand what the incompleteness theorem says. We find that a plausible contention.

The comparison of the various proofs gives much food for thought.

We find it pleasing that the uniqueness of prime factorizations, **PRIME**, the unsolvability of the halting problem, **UNSOLV**, and Gauss's law, **GAUSS**, have "shallow" statements (4, 2, and 3, respectively) but require relatively deep concepts in their proofs (8, 7, and 8). The Church-Rosser theorem, **C-R**, is even more pronounced, with 5 in the statement and 42 in the proof—reflecting the existential functions for constructing the lambda expression alleged to exist. The "computing machine" work, FM8501, FM9001, and Piton, are satisfyingly straightforward—their proofs involve only concepts of the same depth needed to state the theorems. And then there's Gödel's theorem! The existential functions, which construct proofs of inconsistencies from supposed proofs of the undecidable sentence, have depths in the 40,000s because they involve repeated nests of encoders, etc.

Until Gödel's theorem was proved, Gauss's law held the record for the largest number of lemmas in a proof development, with 348. That is dwarfed by Gödel's 1741 and FM9001's 1669. Piton then doubles their figures. Roughly speaking, the three biggest projects run about 11 to 15 lines of text per lemma. However, it was often the case in all three of these projects that the total number of *keystrokes* was dramatically less than the total number of characters in the

final script. The reason is that one often reuses the statements of old theorems by copying them and then editing a few characters. So, for example, in Piton, after having developed a provable statement that the code generator correctly handles the Piton **ADD-NAT** instruction, it was fairly mechanical to generate the corresponding theorems for **ADD-INT** and many other instructions. It should also be noted that line counts are inflated because of the tradition of Nqthm users to use long identifiers, e.g., **ADD-NAT-ONE-WAY-CORRESPONDENCE-R-I**. Because theorem names are seldom typed and new formulas are often created by reusing text already present, the penalty for long identifiers is not as great as might at first appear and is apparently outweighed by the benefits.

Finally, we find the depth of lemma development also satisfying. The prime theorem, Church-Rosser, and FM8501 are all about the same depth, 17 or so levels above the axioms. Gauss's law pushes that to 34, which isn't surprising since it builds on both the prime results and the RSA theorem. But then Gödel's theorem stands 58 levels above Ground Zero. We find it interesting that the Piton compiler proof is not far behind, at 54.

We would like to stress that this attempt to compare the complexities of various theorems and proofs is extremely subjective. Consider FM9001. The theorem essentially says that for "machine states" with certain properties the following two transformations are equivalent: (a) run the machine code n steps; (b) simulate what happens at the gate level for a particular netlist (gate graph) for k steps or "microcycles." To state this fully requires almost 10,000 lines. At 60 lines per page, the FM9001 theorem takes 166 pages *to state* formally.

Our "understandable statement" of the theorem requires only 7,850 lines (131 pages). We consider the function that computes k , the number of microcycles, from the initial state and n , as "existential." That is, we take the position that the reader does not really care what k is, only that there is such a number. Similarly, a person reading this theorem might plausibly be uninterested in the particular gate graph used. One can take the theorem to mean "there is a gate graph that does the right thing." Skipping the lines describing the gate graph reduces the line count to around 4,200 (70 pages).

In order to say what it means to "simulate the netlist" another abstract machine is defined. This machine is essentially an operational semantics for the hardware description language (HDL) used to describe the gate graph. It is reasonable to assume that any designer using a formal verification tool based on this model would invest once in understanding the HDL and then not have to read those definitions defining a familiar language. So one might treat the HDL as primitive (in the same way we treat FM9001 as primitive in the Piton proof

complexity). If one does that the line count is reduced to 1,289 (21 pages).⁴⁵ For that number of lines you get the characterization of the acceptable initial machine states and a definition of the machine code that is implemented by the (uninspected) design. Of course, those lines subtracted from the “understandable statement” must be added to the “lines of supporters” since the details are involved in the proof.

We now turn to the question of the lengths of the formal proofs allegedly checked by Nqthm computations. The difficulty of measuring this stems from the fact that Nqthm does not actually construct a formal proof. Instead, we allege that an Nqthm computation establishes that a formal proof exists. Consider the rewriter, for example. By applying rewrite rules it transforms an input term, t , into an equivalent output term, t' . We regard the rewriter as having proved the theorem (**IMPLIES** **hyps** (**EQUAL** t t')), where **hyps** represents the context in which the rewriting occurred. But the rewriter does not actually construct a formal proof of this theorem, i.e., it does not construct a finite sequence of formulas concluding with the desired theorem and each formula of which is either an axiom or is derived by a rule of inference from previous formulas. But, because of the way we have coded the rewriter, we are personally convinced that such a formal proof could be constructed; indeed, this is the basis of our belief in the soundness of Nqthm. Some parts of the theorem prover, e.g., the linear arithmetic decision procedure and the tautology checker, do computations that are even further removed from the actual construction of a formal proof. Nevertheless, it is possible to instrument the theorem prover so as to estimate the size of the alleged proof. To do this we simply imagine the recoding of the theorem prover to construct formal proofs and accumulate the sizes of the various contributions.

To so instrument Nqthm it was necessary first to decide what to count, i.e., to decide what rules of inference could be used in the virtual formal proof. For example, if the formal proof contains a line equating t and t' , how many formal steps does it take to substitute t' for t in another line? In a formal logic in which equality substitution is handled via an equality axiom for each function symbol, such a substitution takes a number of steps proportional to the depth at which t occurs in the target formula. This is the case, for example, in the formal proofs constructed by HOL [64]. Even in such a logic, one could derive

⁴⁵In [74], Hunt and Brock use the figure 1,112 for the number of lines in the understandable statement. That figure is comparable to our figure of 1,289. The discrepancy of 177 lines is due merely to different decisions as to where to place certain function definitions that are in fact used in different parts of the theorem. For example, if **TRUNC** is used both in the definition of the “existential” microcycle counter and in the definition of the FM9001 itself, is **TRUNC** part of the “understandable statement” or not? This is just another illustration of why it is hard to characterize the size of a theorem in an intuitively meaningful way.

a more sophisticated rule of inference that allowed such a substitution in one step, with the effect of perhaps drastically reducing the size of the formal proofs generated. In this particular case, we opted to allow our formal proofs to use a one-step substitutivity rule because it was most convenient.

Similarly, we decided to count as one step such acts as instantiation, *modus ponens*, tautology recognition, and cross-multiplication and addition of inequalities. Some of these steps are significantly more complicated than generally included in formal proofs, making our proof sizes significantly smaller than they would be in more primitive systems. Consider for example the normalization of a tautological **IF** expression. A naive way to convert that into a formal proof of the **IF** expression would be to lay down a line corresponding to each branch through the **IF** tree, each such line being a propositional tautology. Then one must lay down a line for each interior node of the tree, constructing the desired **IF** expression from the previously generated lines. But in the FM9001 proof, for example, Nqthm recognizes tautological **IF** expressions containing more than 50 **IF**s, meaning that we count as one step a deduction whose formal explanation contains more than 10^9 lines. In any case, having decided what we would count as a single step, we modified the theorem prover so that it would accumulate the number of such steps taken.

But not every step taken by our theorem prover contributes to the final proof; some steps are wasted in fruitless search. Suppose, for example, the rewriter tries to apply a rule with two hypotheses. It must backchain to establish the hypotheses. Suppose the first one is established, at the cost of adding 1,000 lines to the evolving virtual formal proof. But suppose the second one fails; then the whole attempt to apply the rule is abandoned and the theorem prover tries other rules. Even if the theorem prover is ultimately successful in constructing a virtual formal proof, the 1,000 lines described above are not part of that proof. So we must distinguish lines kept from lines generated. A measure of the efficiency of the system (and, primarily, of the user's collection of rewrite rules) is the ratio between these two numbers.

Finally, the virtual proof constructed is not nearly optimal. For example, the same line may be included twice, but this is undetectable in our instrumentation because the actual proof is not kept.

We point out all these qualifications simply to alert the reader, again, to the difficulty of trying to quantify how complicated a proof—even a formal proof—is. We regard the numbers obtained by our instrumented theorem prover as merely indicative of the size of the formal proofs it is checking. We have only run the instrumented machine on one of our twelve theorems, namely, FM9001.

The sum of the kept proof steps in all the formal proofs done for FM9001 is 6,100,315. That is, the size of the formal proof virtually constructed by Nqthm for FM9001 is approximately six million lines. It would be easy to increase this

number by reducing the size of our proof steps, e.g., throwing out one particular use of the tautology recognition rule and replacing it by a more primitive formal proof would add 10^9 steps.⁴⁶

The total number of generated proof steps for FM9001 is 19,165,122. This means that 32% of all the investigated steps were actually kept. Our experience with automated theorem proving leads us to consider this a fairly efficient search and very complimentary to Hunt and Brock, the authors of the data base being used.

Here are some other interesting numbers collected. The total runtime for the FM9001 proof on a Sun Microsystems Sparc-2 running Gnu Common Lisp was roughly 4 hours. The term clausifier was called 80,390 times. The function that determines the type of a term was called 14,699,506 times. The rewriter was called 11,624,455 times.

In closing we would like to reiterate a remark of the Preface: the theorem prover's achievements, as clearly indicated by Table III.1, is, more than anything, a tribute to the people who "led the expeditions" up these peaks.

⁴⁶Is it clear how subjective this game is?

References

1. R. L. Akers. *Strong Static Type Checking for Functional Common LISP*. Ph.D. Th., University of Texas at Austin, 1994. University Microfilms and **ftp://ftp.cs.utexas.edu/pub/boyer/diss/akers.ps.Z**.
2. C. M. Angelo, D. Verkest, L. Claesen, and H. De Man. "On the Comparison of HOL and Boyer-Moore for Formal Hardware Verification". *Formal Methods in System Design: An International Journal* 2, 1 (1993), 45-72.
3. W. Bevier. *A Verified Operating System Kernel*. Ph.D. Th., University of Texas at Austin, 1987. University Microfilms and **ftp://ftp.cs.utexas.edu/pub/boyer/diss/bevier.ps.Z**.
4. W. R. Bevier. A Library for Hardware Verification. Internal Note 57, Computational Logic, Inc., 1988. **ftp://ftp.cs.utexas.edu/pub/boyer/cli-notes/note-057.ps**.
5. W. R. Bevier. "Kit and the Short Stack". *Journal of Automated Reasoning* 5, 4 (1989), 519-530.
6. W. R. Bevier. "Kit: A Study in Operating System Verification". *IEEE Transactions on Software Engineering* 15, 11 (1989), 1382-1396.
7. W. R. Bevier, W. A. Hunt, J S. Moore, and W. D. Young. "Special Issue on System Verification". *Journal of Automated Reasoning* 5, 4 (1989), 409-530.
8. W. R. Bevier and L. Smith. A Mathematical Model of the Mach Kernel: Entities and Relations. Tech. Rept. 88, Computational Logic, Inc., 1993. **ftp://ftp.cs.utexas.edu/pub/boyer/cli-reports/088.ps**.

9. W. R. Bevier and L. Smith. A Mathematical Model of the Mach Kernel: Atomic Actions and Locks. Tech. Rept. 89, Computational Logic, Inc., 1993.
ftp://ftp.cs.utexas.edu/pub/boyer/cli-reports/089.ps.
10. W. R. Bevier and W. D. Young. Machine Checked Proofs of a Byzantine Agreement Algorithm. Tech. Rept. 62, Computational Logic, Inc., 1990.
ftp://ftp.cs.utexas.edu/pub/boyer/cli-reports/062.ps.
11. W. W. Bledsoe. "Splitting and Reduction Heuristics in Automatic Theorem Proving". *Artificial Intelligence* 2 (1971), 55-77.
12. W. W. Bledsoe, R. S. Boyer, and W. Henneman. "Computer Proofs of Limit Theorems". *Artificial Intelligence* 3 (1972), 27-60.
13. D. Borriore, H. Bouamama, D. Deharbe, C. Le Faou, and A. Wahba. HDL-Based Integration of Formal Methods and CAD Tools in the PREVAIL Environment. In *Formal Methods in Computer-Aided Design: First International Conference, FMCAD '96*, M. Srivas and A. Camilleri, Eds., Springer LNCS 1166, 1996, pp. 450-467.
14. D. Borriore, H. Bouamama, and R. Suescun. Validation of the Numeric_Bit Package Using the NQTHM Theorem Prover. Proceedings of the APCHDL'96 Conference, Bangalore, India, 1996.
15. D. Borriore, H. Eveking, L. Pierre. Formal Proofs from HDL Descriptions. In *Fundamentals and Standards in Hardware Description Languages*, J. Mermet, Ed., Kluwer, Dordrecht, 1993, pp. 155-193.
16. D. Borriore, L. Pierre, A. Salem. Formal Verification of VHDL Descriptions in Boyer-Moore: First Results. In *VHDL for Simulation, Synthesis and Formal Proofs*, J. Mermet, Ed., Kluwer, Dordrecht, 1992.
17. D. Borriore, L. Pierre, and P. Salem. "Formal Verification of VHDL Descriptions in the PREVAIL Environment". *IEEE Design and Test Magazine* 9, 2 (1992), 42-56.
18. R. S. Boyer. "Response, Biographical Sketch, and Photo on the Occasion of Receipt of the 1991 AMS Current Award for Automatic Theorem Proving". *Notices of the American Mathematical Society* 38, 5 (1991), 407-408.
19. R. S. Boyer, D. M. Goldschlag, M. Kaufmann, and J S. Moore. Functional Instantiation in First-Order Logic. In *Artificial Intelligence and Mathematical Theory of Computation: Papers in Honor of John McCarthy*, V. Lifschitz, Ed., Academic Press, San Diego, 1991, pp. 7-26.
20. R. S. Boyer, M. W. Green and J S. Moore. The Use of a Formal Simulator to Verify a Simple Real Time Control Program. In *Beauty is Our Business: A Birthday Salute to Edsger W. Dijkstra*, W. H. J. Feijen, A. J. M. van Gasteren, D. Gries, and J. Misra, Eds., Springer Texts and Monographs in Computer Science, 1990, pp. 54-66.

21. R. S. Boyer, M. Kaufmann, and J S. Moore. "The Boyer-Moore Theorem Prover and Its Interactive Enhancement". *Computers and Mathematics with Applications* 29, 2 (1995), 27-62.
22. R. S. Boyer and J S. Moore. "Proving Theorems about LISP Functions". *Journal of the ACM* 22, 1 (1975), 129-144.
23. R. S. Boyer and J S. Moore. A Lemma Driven Automatic Theorem Prover for Recursive Function Theory. Proceedings of the 5th International Joint Conference on Artificial Intelligence, 1977, pp. 511-519.
24. R. S. Boyer and J S. Moore. *A Computational Logic*. Academic Press, New York, 1979.
25. R. S. Boyer and J S. Moore. Metafunctions: Proving Them Correct and Using Them Efficiently as New Proof Procedures. In *The Correctness Problem in Computer Science*, R. S. Boyer and J S. Moore, Eds., Academic Press, London, 1981.
26. R. S. Boyer and J S. Moore. A Verification Condition Generator for FORTRAN. In *The Correctness Problem in Computer Science*, R. S. Boyer and J S. Moore, Eds., Academic Press, London, 1981.
27. R. S. Boyer and J S. Moore. "Proof Checking the RSA Public Key Encryption Algorithm". *American Mathematical Monthly* 91, 3 (1984), 181-189.
28. R. S. Boyer and J S. Moore. A Mechanical Proof of the Turing Completeness of Pure Lisp. In *Automated Theorem Proving: After 25 Years*, W. W. Bledsoe and D. W. Loveland, Eds., American Mathematical Society, Providence, RI, 1984, pp. 133-167.
29. R. S. Boyer and J S. Moore. "A Mechanical Proof of the Unsolvability of the Halting Problem". *Journal of the ACM* 31, 3 (1984), 441-458.
30. R. S. Boyer and J S. Moore. "The Addition of Bounded Quantification and Partial Functions to A Computational Logic and Its Theorem Prover". *Journal of Automated Reasoning* 4, 2 (1988), 117-172.
31. R. S. Boyer and J S. Moore. Integrating Decision Procedures into Heuristic Theorem Provers: A Case Study with Linear Arithmetic. In *Machine Intelligence 11*, J. E. Hayes, D. Michie, and J. Richards, Eds., Oxford University Press, Oxford, 1988.
32. R. S. Boyer and J S. Moore. MJRTY - A Fast Majority Vote Algorithm. In *Automated Reasoning: Essays in Honor of Woody Bledsoe*, R. S. Boyer, Ed., Kluwer, Dordrecht, 1991, pp. 105-117.
33. R. S. Boyer and Y. Yu. Automated Correctness Proofs of Machine Code Programs for a Commercial Microprocessor. In *Automated Deduction -- CADE-11*, D. Kapur, Ed., Springer LNCS 607, 1992, pp. 416-430.

34. R. S. Boyer and Y. Yu. A Formal Specification of Some User Mode Instructions for the Motorola 68020. Tech. Rept. TR-92-04, Computer Sciences Department, University of Texas at Austin, 1992. **ftp://ftp.cs.utexas.edu/pub/techreports/tr92-04.ps.Z**.
35. R. S. Boyer and Y. Yu. "Automated Correctness Proofs of Machine Code Programs for a Widely Used Microprocessor". *Journal of the ACM* 43, 1 (1996), 166-192.
36. J. D. Bright, G. F. Sullivan, and G. M. Mason. A Formally Verified Sorting Certifier. To appear in *IEEE Transactions on Computers*.
37. B. Brock, W. A. Hunt, and M. Kaufmann. The FM9001 Microprocessor Proof. Tech. Rept. 86, Computational Logic, Inc., 1994. **ftp://ftp.cs.utexas.edu/pub/boyer/cli-reports/086.ps**.
38. B. Brock, W. A. Hunt, and W. D. Young. Introduction to a Formally Defined Hardware Description Language. In *Theorem Provers in Circuit Design*, V. Stavridou, T. Melham, and R. Boute, Eds., North-Holland, Amsterdam, 1992, pp. 3-35.
39. B. Brock, M. Kaufmann, and J. S. Moore. ACL2 Theorems about Commercial Microprocessors. In *Formal Methods in Computer-Aided Design: First International Conference, FMCAD '96*, M. Srivas and A. Camilleri, Eds., Springer LNCS 1166, 1996, pp. 275-293.
40. A. Bronstein. *MLP: String-Functional Semantics and Boyer-Moore Mechanization for the Formal Verification of Synchronous Circuits*. Ph.D. Th., Stanford University, 1989.
41. A. Bronstein and C. Talcott. String-Functional Semantics for Formal Verification of Synchronous Circuits, Report No. STAN-CS-88-1210. Computer Science Department, Stanford University, 1988.
42. A. Bronstein and C. Talcott. Formal Verification of Synchronous Circuits Based on String-Functional Semantics: The 7 Paillet Circuits in Boyer-Moore. C-Cube 1989 Workshop on Automatic Verification Methods for Finite State Systems, 1989, pp. 317-333. Springer LNCS 407.
43. A. Bronstein and C. Talcott. Formal Verification of Pipelines Based on String-Functional Semantics. IFIP International Workshop on Applied Formal Methods for Correct VLSI Design, Leuven, Belgium, 1989.
44. R. Burstall. "Proving Properties of Programs by Structural Induction". *The Computer Journal* 12, 1 (1969), 41-48.
45. M. Carranza and W. D. Young. Verification of a Fuzzy Controller. Tech. Rept. 82, Computational Logic, Inc., 1992. **ftp://ftp.cs.utexas.edu/pub/boyer/cli-reports/082.ps**.

46. M. Carranza and W. D. Young. A Fuzzy Controller: Theorems and Proofs about its Dynamic Behavior. Tech. Rept. 91, Computational Logic, Inc., 1993. <ftp://ftp.cs.utexas.edu/pub/boyer/cli-reports/091.ps>.
47. K. M. Chandy and J. Misra. *Parallel Program Design: A Foundation*. Addison-Wesley, Reading, MA, 1988.
48. J. Cook and S. Subramanian. A Formal Semantics for C in Nqthm. Tech. Rept. 517D, Trusted Information Systems, 1994.
49. J. R. Cowles. Using NQTHM to Verify Insert, Search, and Traversal Functions for Search Trees. Internal Note 241, Computational Logic, Inc., 1991. <ftp://ftp.cs.utexas.edu/pub/boyer/cli-notes/note-241.ps>.
50. J. R. Cowles. Meeting a Challenge of Knuth. Internal Note 286, Computational Logic, Inc., 1993. <ftp://ftp.cs.utexas.edu/pub/boyer/cli-notes/note-286.txt>.
51. Cygnet Publishing Technologies. *Scribe: Document Production Software User Manual*. 1991. 355 Fifth Avenue, Suite 1515, Pittsburgh, PA 15222-2407.
52. B. L. Di Vito. *Verification of Communications Protocols and Abstract Process Models*. Ph.D. Th., University of Texas at Austin, 1982. University Microfilms.
53. A. Flatau. *A Verified Implementation of an Applicative Language with Dynamic Storage Allocation*. Ph.D. Th., University of Texas at Austin, 1992. University Microfilms and <ftp://ftp.cs.utexas.edu/pub/boyer/diss/flatau.ps.Z>.
54. R. Floyd. Assigning Meanings to Programs. In *Mathematical Aspects of Computer Science, Proceedings of Symposia in Applied Mathematics*, American Mathematical Society, Providence, RI, 1967, pp. 19-32.
55. M. Gardner. "Mathematical Recreation Column". *Scientific American* 203, 2 (August, 1960), 149-154.
56. G. Gentzen. New Version of the Consistency Proof for Elementary Number Theory. In *The Collected Papers of Gerhard Gentzen*, M. E. Szabo, Ed., North-Holland, Amsterdam, 1969, pp. 132-213.
57. S. M. German and Y. Wang. Formal Verification of Parameterized Hardware Designs. International Conference on Computer Design, 1985. Reprinted in [153].
58. N. Gilbreath. "Magnetic Colors". *The Linking Ring* 38, 5 (1958), 60.

59. P. Y. Gloess. An Experiment with the Boyer-Moore Theorem Prover: A Proof of the Correctness of a Simple Parser of Expressions. In *5th Conference on Automated Deduction*, Springer LNCS 87, 1980, pp. 154-169.
60. D. M. Goldschlag. Mechanizing Unity. In *Programming Concepts and Methods*, M. Broy and C. B. Jones, Eds., North-Holland, Amsterdam, 1990.
61. D. M. Goldschlag. "Mechanically Verifying Concurrent Programs with the Boyer-Moore Prover". *IEEE Transactions on Software Engineering* 16 (September 1990), 1005-1023.
62. D. M. Goldschlag. *Mechanically Verifying Concurrent Programs*. Ph.D. Th., University of Texas at Austin, 1992. University Microfilms and `ftp://ftp.cs.utexas.edu/pub/boyer/diss/goldschlag.ps.Z`.
63. D. I. Good, A. Siebert, and W. D. Young. Middle Gypsy 2.05 Definition. Tech. Rept. 59, Computational Logic, Inc., 1990. `ftp://ftp.cs.utexas.edu/pub/boyer/cli-reports/059.ps`.
64. M. J. C. Gordon and T. F. Melham. *Introduction to HOL: A Theorem Proving Environment for Higher Order Logic*. Cambridge University Press, 1993.
65. C. A. R. Hoare. "An Axiomatic Basis for Computer Programming". *Communications of the ACM* 12, 10 (1969), 576-583.
66. C.-H. Huang and C. Lengauer. "The Automated Proof of a Trace Transformation for a Bitonic Sort". *Theoretical Computer Science* 1 (1986), 261-284.
67. G. Huet. The Gallina Specification Language: A Case Study. In *Proceedings of 12th FST/TCS Conference, New Delhi*, R. Shyamasundar, Ed., Springer LNCS 652, 1992, pp. 229-240.
68. G. Huet. Axiomatisations, Proofs, and Formal Specifications of Algorithms: Commented Case Studies in the Coq Proof Assistant. In *Logic of Computation: NATO ASI Series F: Computer and Systems Sciences, Vol. 157*, H. Schwichtenberg, Ed., Springer, 1997.
69. G. Huet and D. Oppen. Equations and Rewrite Rules: A Survey. In *Formal Languages: Perspectives and Open Problems*, R. Book, Ed., Academic Press, San Diego, 1980.
70. W. A. Hunt. *FM8501: A Verified Microprocessor*. Ph.D. Th., University of Texas at Austin, 1985. University Microfilms. Published as [73].
71. W. A. Hunt. The Mechanical Verification of a Microprocessor Design. In *From HDL Descriptions to Guaranteed Correct Circuit Designs*, D. Borrione, Ed., North-Holland, Amsterdam, 1987, pp. 89-132. Reprinted in [153].

72. W. A. Hunt. "Microprocessor Design Verification". *Journal of Automated Reasoning* 5, 4 (1989), 429-460.
73. W. A. Hunt. *FM8501: A Verified Microprocessor*. Springer LNCS 795, 1994.
74. W. A. Hunt and B. Brock . "A Formal HDL and Its Use in the FM9001 Verification". *Philosophical Transactions of the Royal Society, Series A* 339 (1992). Reprinted in *Mechanized Reasoning and Hardware Design*, C. Hoare and M. Gordon, Eds., Prentice-Hall, Englewood Cliffs, pp. 35-47, 1992.
75. M. Kaufmann. A User's Manual for an Interactive Enhancement to the Boyer-Moore Theorem Prover. Tech. Rept. 19, Computational Logic, Inc., May, 1988. **ftp://ftp.cs.utexas.edu/pub/boyer/cli-reports/019.ps**.
76. M. Kaufmann. Addition of Free Variables to an Interactive Enhancement of the Boyer-Moore Theorem Prover. Tech. Rept. 42, Computational Logic, Inc., 1990. **ftp://ftp.cs.utexas.edu/pub/boyer/cli-reports/042.ps**.
77. M. Kaufmann. An Instructive Example for Beginning Users of the Boyer-Moore Theorem Prover. Internal Note 185, Computational Logic, Inc., 1990. **ftp://ftp.cs.utexas.edu/pub/boyer/cli-notes/note-185.ps**.
78. M. Kaufmann. An Integer Library for Nqthm. Internal Note 182, Computational Logic, Inc., 1990. **ftp://ftp.cs.utexas.edu/pub/boyer/cli-notes/note-182.ps**.
79. M. Kaufmann. An Example in Nqthm: Ramsey's Theorem. Internal Note 100, Computational Logic, Inc., 1991, **ftp://ftp.cs.utexas.edu/pub/boyer/cli-notes/note-100.ps**.
80. M. Kaufmann. "Generalization in the Presence of Free Variables: A Mechanically-Checked Proof for One Algorithm". *Journal of Automated Reasoning* 7, 1 (March 1991), 109-158.
81. M. Kaufmann. A Simple Example for Nqthm: Modeling Locking. Internal Note 216, Computational Logic, Inc., 1991. **ftp://ftp.cs.utexas.edu/pub/boyer/cli-notes/note-216.ps**.
82. M. Kaufmann. Response to FM91 Survey of Formal Methods: Nqthm and Pc-Nqthm. Tech. Rept. 75, Computational Logic, Inc., 1992. **ftp://ftp.cs.utexas.edu/pub/boyer/cli-reports/075.ps**.
83. M. Kaufmann. "An Extension of the Boyer-Moore Theorem Prover to Support First-Order Quantification". *Journal of Automated Reasoning* 9, 3 (December 1992), 355-372.

- 84.** M. Kaufmann and J. S. Moore. "An Industrial Strength Theorem Prover for a Logic Based on Common Lisp". *IEEE Transactions on Software Engineering* 23, 4 (April 1997), 203-213.
- 85.** M. Kaufmann and P. Pecchiari. "Interaction with the Boyer-Moore Theorem Prover: A Tutorial Study Using the Arithmetic-Geometric Mean Theorem". *Journal of Automated Reasoning* 16, 1-2 (1996), 181-222.
- 86.** B. W. Kernighan and D. M. Ritchie. *The C Programming Language, Second Edition*. Prentice-Hall, Englewood Cliffs, 1988.
- 87.** M. Kim. *On Automatically Generating and Using Examples in a Computational Logic System*. Ph.D. Th., University of Texas at Austin, 1986. University Microfilms and <ftp://ftp.cs.utexas.edu/pub/boyer/diss/kim.ps.Z>.
- 88.** D. E. Knuth and P. Bendix. Simple Word Problems in Universal Algebras. In *Computational Problems in Abstract Algebras*, J. Leech, Ed., Pergamon Press, Oxford, 1970, pp. 263-297.
- 89.** K. Kunen. "A Ramsey Theorem in Boyer-Moore Logic". *Journal of Automated Reasoning* 15 (1995), 217-235.
- 90.** K. Kunen. Non-constructive Computational Mathematics. To appear in *Journal of Automated Reasoning*.
- 91.** L. Lamport. *LaTeX: A Document Preparation System*. Addison-Wesley, Reading, MA, 1986.
- 92.** C. Lengauer. "On the Role of Automated Theorem Proving in the Compile-Time Derivation of Concurrency". *Journal of Automated Reasoning* 1, 1 (1985), 75-101.
- 93.** C. Lengauer. "A View of Automated Proof Checking and Proving". *Annals of Operations Research* 16 (1988), 61-80.
- 94.** C. Lengauer and C.-H. Huang. A Mechanically Certified Theorem about Optimal Concurrency of Sorting Networks. Proceedings of the 13th Annual ACM Symposium on Principles of Programming Languages, 1986, pp. 307-317.
- 95.** J. McCarthy. *The Lisp Programmer's Manual*. M.I.T. Computation Center, 1960.
- 96.** J. McCarthy. "Recursive Functions of Symbolics Expressions and their Computation by Machine". *Communications of the ACM* 3, 4 (1960), 184-195.
- 97.** J. McCarthy. Computer Programs for Checking Mathematical Proofs. In *Recursive Function Theory: Proceedings of a Symposium in Pure Mathematics*, American Mathematical Society, Providence, RI, 1962, pp. 219-227.

- 98.** J. McCarthy. Towards a Mathematical Science of Computation. Proceedings of IFIP Congress, Amsterdam, 1962, pp. 21-28.
- 99.** J. McCarthy. A Basis for a Mathematical Theory of Computation. In *Computer Programming and Formal Systems*, P. Braffort and D. Hershberg, Eds., North-Holland, Amsterdam, 1963.
- 100.** J. McCarthy, et al. *LISP 1.5 Programmer's Manual*. The MIT Press, 1965.
- 101.** W. McCune. A Proof-Checker for Resolution/Paramodulation Proofs. Mathematics and Computer Science Division, Argonne National Laboratory, 1995. <ftp://info.mcs.anl.gov/pub/Otter/QED/check-4.tar.gz>.
- 102.** J S. Moore. "A Mechanical Proof of the Termination of Takeuchi's Function". *Information Processing Letters* 9, 4 (1979), 176-181.
- 103.** J S. Moore. "Response, Biographical Sketch, and Photo on the Occasion of Receipt of the 1991 AMS Current Award for Automatic Theorem Proving". *Notices of the American Mathematical Society* 38, 5 (1991), 408-410.
- 104.** J S. Moore. Mechanically Verified Hardware Implementing an 8-Bit Parallel IO Byzantine Agreement Processor. Tech. Rept. 69, Computational Logic, Inc., 1991. <ftp://ftp.cs.utexas.edu/pub/boyer/cli-reports/069.ps>.
- 105.** J S. Moore. "A Formal Model of Asynchronous Communication and Its Use in Mechanically Verifying a Biphase Mark Protocol". *Formal Aspects of Computing* 6, 1 (1994), 60-91.
- 106.** J S. Moore. *Piton: A Mechanically Verified Assembly-Level Language*. Kluwer, Dordrecht, 1996.
- 107.** Motorola, Inc. *MC68020 32-bit Microprocessor User's Manual*. 1989. Prentice-Hall, Englewood Cliffs.
- 108.** M. Nagayama and C. Talcott. An NQTHM Mechanization of "An Exercise in the Verification of Multi-Process Programs". Tech. Rept. STAN-CS-91-1370, Computer Science Department, Stanford University, 1991.
- 109.** F. Nicoli and L. Pierre. Formal Verification of Behavioural VHDL Specifications : A Case Study. Proceedings of the International Conference EURO-DAC with EURO-VHDL, 1994. IEEE Computer Society Press.
- 110.** L. Pierre. VHDL Description and Formal Verification of Systolic Multipliers. In *Computer Hardware Description Languages and their Applications*, D. Agnew and L. Claesen, Eds., North-Holland, Amsterdam, 1993.

- 111.** L. Pierre. Describing and Verifying Synchronous Circuits with the Boyer-Moore Theorem Prover. In *Correct Hardware Design and Verification Methods*, P. Camurati and H. Eveking, Eds., Springer LNCS 987, 1995.
- 112.** L. Pierre. An Automatic Generalization Method for the Inductive Proof of Replicated and Parallel Architectures. In *Theorem Provers in Circuit Design*, R. Kumar and T. Kropf, Eds., Springer LNCS 901, 1995.
- 113.** L. Pierre. VHDL Description, Boyer-Moore Specification and Formal Verification of a Parallel System for Finding a Maximum. Proceedings of a Workshop on Formal Methods for Parallel Programming : Theory and Applications, 1997. FMPPTA 97.
- 114.** J. D. Ramsdell. The Tail Recursive SECD Machine. <ftp://ftp.cs.utexas.edu/pub/boyer/nqthm/trsecd/trsecd.html>.
- 115.** S. Read. *Formal Methods for VLSI Design*. Ph.D. Th., University of Manchester Institute of Science and Technology, 1994.
- 116.** S. Read and M. Edwards. A Formal Semantics of VHDL in Boyer-Moore Logic. Proceedings of the International Conference on Concurrent Engineering and Electronic Design Automation, Poole, Great Britain, 1994, pp. 395-400.
- 117.** R. Rivest, A. Shamir, and L. Adleman. "A Method for Obtaining Digital Signatures and Public-Key Cryptosystems". *Communications of the ACM* 21, 2 (1978), 120-126.
- 118.** H. Rogers, Jr. *Theory of Recursive Functions and Effective Computability*. McGraw-Hill, New York, 1967.
- 119.** D. M. Russinoff. A Mechanical Proof of Wilson's Theorem. Master Th., University of Texas at Austin, 1983.
- 120.** D. M. Russinoff. "An Experiment with the Boyer-Moore Theorem Prover: A Proof of Wilson's Theorem". *Journal of Automated Reasoning* 1, 2 (1985), 121-139.
- 121.** D. M. Russinoff. "A Mechanical Proof of Quadratic Reciprocity". *Journal of Automated Reasoning* 8, 1 (1992), 3-21.
- 122.** D. M. Russinoff. "A Mechanically Verified Incremental Garbage Collector". *Formal Aspects of Computing* 6 (1994), 359-390.
- 123.** D. M. Russinoff. Specification and Verification of Gate-Level VHDL Models of Synchronous and Asynchronous Circuits. In *Specification and Validation Methods*, E. Boerger, Ed., Oxford University Press, 1995.
- 124.** D. M. Russinoff. "A Formalization of a Subset of VHDL". *Formal Methods in System Design* 7 (1995), 7-25. In special issue on VHDL Semantics.

- 125.** A. Salem and D. Borriore. Formal Semantics of VHDL Timing Constructs. In *VHDL for Simulation, Synthesis and Formal Proofs of Hardware*, J. Mermet, Ed., Kluwer, Dordrecht, 1992, pp. 195-206.
- 126.** N. Shankar. "Towards Mechanical Metamathematics". *Journal of Automated Reasoning* 1, 4 (1985), 407-434.
- 127.** N. Shankar. *Proof Checking Metamathematics*. Ph.D. Th., University of Texas at Austin, 1986. Published as [129].
- 128.** N. Shankar. "A Mechanical Proof of the Church-Rosser Theorem". *Journal of the ACM* 35, 3 (1988), 475-522.
- 129.** N. Shankar. *Metamathematics, Machines, and Goedel's Proof*. Cambridge University Press, 1994.
- 130.** J. R. Shoenfield. *Mathematical Logic*. Addison-Wesley, Reading, MA, 1967.
- 131.** G. L. Steele, Jr. *Common Lisp The Language*. Digital Press, 30 North Avenue, Burlington, MA 01803, 1984.
- 132.** G. L. Steele, Jr. *Common Lisp The Language, Second Edition*. Digital Press, 30 North Avenue, Burlington, MA 01803, 1990.
- 133.** S. Subramanian. *A Mechanized Framework for Specifying Problem Domains and Verifying Plans*. Ph.D. Th., University of Texas at Austin, 1993. University Microfilms and **ftp://ftp.cs.utexas.edu/pub/boyer/diss/subramanian.ps.Z**.
- 134.** S. Subramanian. "An Interactive Solution to the $n \times n$ Mutilated Checkboard Problem". *Journal of Logic and Computation* 6, 4 (1996), 573-598.
- 135.** S. Subramanian and J. Cook. Mechanical Verification of C Programs. Proceedings of the ACM SIGSOFT Workshop on Formal Methods in Software Practice, 1996, pp. 20-30.
- 136.** S. Subramanian and J. Cook. Automatic Verification of Object Code Against Source Code. Proceedings of the Eleventh Annual Conference on Computer Assurance (COMPASS'96), 1996, pp. 46-55.
- 137.** W. Teitelman. *INTERLISP Reference Manual*. Xerox Palo Alto Research Center, 3333 Coyote Hill Road, Palo Alto, CA, 1978.
- 138.** D. A. Turner. "A New Implementation Technique for Applicative Languages". *Software -- Practice and Experience* 9 (1979), 31-49.
- 139.** D. Verkest, L. Claesen, and H. De Man. On the Use of the Boyer-Moore Theorem Prover for Correctness Proofs of Parameterized Hardware Modules. In *Formal VLSI Specification and Synthesis*, L. Claesen, Ed., Elsevier, Amsterdam, 1990, pp. 99-116.

140. D. Verkest, L. Claesen, and H. De Man. Correctness Proofs of Parameterized Hardware Modules in the Cathedral-II Synthesis Environment. Proceedings of the European Design Automation Conference, EDAC 90, 1990, pp. 62-66.
141. D. Verkest, L. Claesen, and H. De Man. A Proof of the Non-Restoring Division Algorithm and its Implementation on the Cathedral-II ALU. In *Designing Correct Circuits*, J. Staunstrup and R. Sharp, Ed., Elsevier, Amsterdam, 1992, pp. 173-192.
142. D. Verkest, L. Claesen, and H. De Man. "A Proof of the Non-Restoring Division Algorithm and its Implementation on an ALU". *Journal of Formal Methods in System Design* 4, 1 (1994), 5-31.
143. D. Verkest, J. Vandenbergh, L. Claesen, and H. De Man. A Description Methodology for Parameterized Modules in the Boyer-Moore Logic. In *Theorem Provers in Circuit Design: Theory, Practice and Experience*, V. Stavridou, T. Melham, and R. Boute, Ed., Elsevier, Amsterdam, 1992, pp. 37-57.
144. Hao Wang. "Toward Mechanical Mathematics". *IBM Journal* 4 (1960), 2-22.
145. D. Weber-Wulff. "Proof Movie : A Proof with the Boyer-Moore Prover". *Formal Aspects of Computing* 5 (1993), 121-151.
146. D. Weber-Wulff. *Contributions to Mechanical Proofs of Correctness for Compiler Front-Ends*. Ph.D. Th., University of Kiel, Germany, Department of Computer Science, 1997. Technical Report IfI 9707.
147. M. Wilding. A Mechanically-Checked Correctness Proof of a Floating-Point Search Program. Tech. Rept. 56, Computational Logic, Inc., 1990. **ftp://ftp.cs.utexas.edu/pub/boyer/cli-reports/056.ps**.
148. M. Wilding. "Proving Matijasevich's Lemma with a Default Arithmetic Strategy". *Journal of Automated Reasoning* 7, 3 (1991), 439-446.
149. M. Wilding. A Proved Application with Simple Real Time Properties. Tech. Rept. 78, Computational Logic, Inc., 1992. **ftp://ftp.cs.utexas.edu/pub/boyer/cli-reports/078.ps**.
150. M. Wilding. A Mechanically Verified Application for a Mechanically Verified Environment. In *Computer-Aided Verification -- CAV '93*, C. Courcoubetis, Ed., Springer LNCS 697, 1993.
151. M. Wilding. *Machine-Checked Real-Time System Verification*. Ph.D. Th., University of Texas at Austin, May 1996. University Microfilms, **ftp://ftp.cs.utexas.edu/pub/boyer/diss/wilding.ps.Z**, and **ftp://ftp.cs.utexas.edu/pub/boyer/diss/wilding-diss-events.tar.gz**.

152. M. Wilding. A Detailed Processor Model for Verification of Real-Time Applications. In *2nd IFAC Workshop on Safety and Reliability in Emerging Control Technologies*, T. Hilburn, G. Suski, and J. Zalewski, Eds., Pergamon/Elsevier Science, 1996.
153. M. Yoeli, (Ed.). *Formal Verification of Hardware Design*. IEEE Computer Society Press Tutorial, 1990.
154. W. D. Young. *A Verified Code-Generator for a Subset of Gypsy*. Ph.D. Th., University of Texas at Austin, 1988. University Microfilms and **ftp://ftp.cs.utexas.edu/pub/boyer/diss/young.ps.Z**.
155. W. D. Young. A Simple Expression Compiler: A Programming and Proof Example for an Nqthm Course. Internal Note 210, Computational Logic, Inc., 1990. **ftp://ftp.cs.utexas.edu/pub/boyer/cli-notes/note-210.ps**.
156. W. D. Young. Verifying the Interactive Convergence Clock Synchronization Algorithm Using the Boyer-Moore Theorem Prover. Tech. Rept. 77, Computational Logic, Inc., 1991. **ftp://ftp.cs.utexas.edu/pub/boyer/cli-reports/077.ps**.
157. W. D. Young. Modeling and Verification of a Simple Real-Time Gate Controller. In *Applications of Formal Methods*, M. Hinchey and J. Bowen, Eds., Prentice-Hall, London, 1994. **ftp://ftp.cs.utexas.edu/pub/boyer/cli-reports/093.ps**.
158. Y. Yu. "Computer Proofs in Group Theory". *Journal of Automated Reasoning* 6, 3 (1990).
159. Y. Yu. *Automated Proofs of Object Code for a Widely Used Microprocessor*. Ph.D. Th., University of Texas at Austin, 1992. University Microfilms and **ftp://ftp.cs.utexas.edu/pub/boyer/diss/yu.ps.Z**.

Index

\oplus 55
 \otimes 131

#-comment 144
#-sequence 142
#B (binary notation) 30, 142
#O (octal notation) 30, 142
#X (hexadecimal notation) 31, 142
#|-occurrence 144

' (single quote) 32, 142

*1*FALSE 159, 374
*1*fn 337
*1*QUOTE 159, 162, 374
*1*TRUE 159, 374
COMPILE-FUNCTIONS-FLG 317
DEFAULT-NQTHM-PATH 343

, (comma) 142
, (in backquote notation) 34
,. (comma dot) 143
,@ (comma at-sign) 143
,@ (in backquote notation) 34

. (dot) 142

/ (application of a variable substitution) 129

91-function 95

A

A-D-SEQUENCE 458
A-D-SEQUENCEP 458
A/D sequence 154
ABBREV 473
ABBREV (R-LOOP special form) 375
Abbreviated **FOR** 62, 178
ABBREVIATED-FOR-BODY 466
ABBREVIATED-FOR-OP 466
ABBREVIATED-FOR-RANGE 466
ABBREVIATED-FOR-VAR 466
ABBREVIATED-FOR-WHEN 466
ABBREVIATED-FORP 467
Abbreviation 121, 125, 129, 130, 138, 164, 178
Abbreviation rules 268
Aborting commands 226, 295
Abs 422
Accessor 23, 28, 132
ACCUMULATED-PERSISTENCE 225, 297
ack.events 429
Ackermann's function 6
Acute accent (') 142
Ada 11
ADD-AXIOM 224, 298
ADD-ELEMENT-TO-TOP 453
ADD-SHELL 224, 298
ADD-TO-SET 52, 167
ADD-TO-SET (quantifier operation) 61, 177
ADD1 25, 135, 476
ADD1-NESTP 473

Addition 136
 Admissibility (of definition) 181
 Admissibility (of shell invocation) 133
AEXSYN 473
 AFOSR xix
 AKCL 414
 Akers, Larry 265
 Alist 37, 45
 Alist (standard alist) 175
 All (\forall) xxiii, 80, 435
ALL-BASE-N-DIGIT-CHARACTERS 443
ALL-BUT-LAST 444
ALL-DEPENDENTS (**DATA-BASE** query) 322
ALL-SATISFY 462
ALL-SUPPORTERS (**DATA-BASE** query) 323
ALL-UPPER-ALPHABETICS-DIGITS-OR-SIGNS 460
ALL-VARS 467
ALPHABETIC-INSERT 467
ALPHABETIC-LESSP 467
ALPHABETIC-LESSP1 467
ALPHABETIZE 467
alternating.events 423
ALWAYS (quantifier operation) 61, 177
amax.events 431
 Ancestor 345
ANCESTORS (**DATA-BASE** query) 323
 Ancestral 345
AND 22, 31, 130
APP 93
app-c-d-e.events 426
app-f.events 426
APPEND 44, 52, 167, 193, 196
APPEND (quantifier operation) 61, 177
 Applicability (of a rule) 279
APPLY\$ 57, 174
APPLY-SUBR 53, 56, 135, 175, 477, 479, 480, 482
 Argument list (of a **LAMBDA** expression) 183
 Arguments 125
 Arithmetic 6, 25, 51, 83, 135, 166, 200, 270, 476
arithmetic-geometric-mean.events 430
 Arity 122
ARITY 463
 Arity (of a **LAMBDA** expression) 183
ARITY-ALIST 463
 ARPA xix, xxv
 Artificial intelligence 71
ASCII 442
ASCII-DIGITS-AND-SIGNS 460

ASCII-LIST 442
ASCII-TABLE 440
ASCII-UPPER-ALPHABETICS 460
asm.events 432
 Assembler verification 12, 486
 Assembly language 8
ASSOC 45, 52, 168
 Association list 37, 45
 Associativity (rule of inference) 128
async18.events 423
 Asynchronous communication 8
 Atomic formula 124
 Atomic symbol 136, 193, 479
AXIOM 224, 300
 Axiomatic acts 126
 Axioms (of a history) 127
 Axioms added (by an axiomatic act) 126
 Axioms added (by definition) 182
 Axioms added (by shell invocation) 134

B

Backquote expansion 151
 Backquote notation 34
 Backquote token (\backslash) 142
 Backquote token tree 143
BACKQUOTE-EXPANSION 456
BACKQUOTE-SETTING 301
BACKQUOTE-TOKEN 445
BACKQUOTE-TOKEN-TREE 447
 Bags 6
bags.events 430
 Balanced number signs 144
 Base case 180
 Base function 23, 28, 132
 Base functions (of a history) 132
 Base n digit character 141
 Base n digit sequence 141
 Base n signed value 141
 Base n value 141
BASE-N-DIGIT-CHARACTER 443
BASE-N-DIGIT-SEQUENCE 443
BASE-N-DIGIT-VALUE 443
BASE-N-SIGNED-VALUE 444
BASE-N-VALUE 443
 Bendix, P. 400
 Berkeley C string library 11, 432
 Bevier, Bill xviii, xxii, xxiv, 5, 11, 12, 425, 430
 Bevier-mode 348
big-add.events 427
 Big-delta 422
 Binary notation 30, 142
 Binary search 11

- Binary tree 38, 45, 136, 478
- Binomial theorem 8
- binomial.events** 423
- Biphase mark 8
- Bitonic sort 9
- Bledsoe, Woody 4
- Block-not-in-use 422
- Body 40
- BODY** 53, 57, 59, 174, 175, 176
- Body (of a **LAMBDA** expression) 183
- Boolean-sample xxiv, 421
- BOOT-STRAP** 224, 303
- Borrione, Dominique 6, 11
- Bound variable (in functional substitutions) 184
- Bound variable (in **LAMBDA** expressions) 184
- Bounded quantification 61, 177
- Boxer, Alan xxiv
- Break character 145
- BREAK-LEMMA** 225, 305
- BREAK-REWRITE** 312
- Bridge lemmas 406
- Bright, Jon 6, 12
- British Petroleum xix
- Brock, Bishop xxii, xxiv, 6, 11, 427, 486
- Bronstein, Alex xxiv, 6, 11, 426
- bronstein/*.events** 426
- bsearch.events** 432
- Burstall, Rod 206
- C
- C (programming language) 11
- CAAR** 31, 154, 157
- CADR** 31, 154, 157
- CADRn** 442
- C...A/D...R**, etc. 31, 154, 157
- Call 125
- Called in 125
- Canonical constants 137
- CAR** 26, 136, 478
- CAR-CDR-NEST** 458
- CAR-CDR-SYMBOLP** 458
- CAR/CDR** nest 154
- CAR/CDR** symbol 127
- Card trick 6
- Carranza, Miren 6, 10
- CASE** 31, 127, 156
- Case splits, how to avoid 408
- Cases (of an induction) 180
- CDR** 26, 136, 478
- CDR** nest 175
- CDRn** 442
- CH** 225, 314
- Chandy, Mandy 9
- Check point 252, 259
- Chou, S.C. xviii
- CHRONOLOGY** 316
- Church-Rosser theorem 7, 118, 485
- church-rosser.events** 431
- Citizen 319
- Claesen, Luc 6, 11
- Classes (of rules) 275, 381
- Clause 264
- CLI short stack 12
- CLOSE-PAREN** 448
- Cohen, Ernie xviii
- COLLECT** (quantifier operation) 62, 177
- Comma at-sign token (,@) 143
- Comma dot token (,.) 143
- Comma escape token tree 144
- Comma token (,) 142
- COMMA-AT-SIGN-TOKEN** 445
- COMMA-DOT-TOKEN** 445
- COMMA-ESCAPE-TOKEN-TREE** 447
- COMMA-TOKEN** 445
- Command file 257
- Commands (summary) 224
- Comment 21, 145
- COMMENT** 316
- Common Lisp 414
- Communication protocols 8
- Commutative rules 280, 401
- *COMPILE-FUNCTIONS-FLG*** 317
- COMPILE-NQTHM** 415
- COMPILE-UNCOMPILED-DEFNS** 317
- Compiler verification 12
- Compiling 8, 317, 338, 341, 343, 415, 484
- Complexity (of proofs) 483
- Compound recognizer formula 285
- Compound recognizer rule 277, 284, 285
- Computational logic 3
- Computational Logic, Inc. xix, xxv
- Computing (in the logic) 137
- Computing machines 104
- CONCAT** 461
- Concurrency 9
- COND** 31, 127, 155
- CONS** 26, 136, 478
- CONS-ADD1-NESTP** 473
- CONSP** 36
- CONSTITUENT** 447
- CONSTRAIN** 224, 317
- Constraining new symbols 184
- Constructor 23, 28, 132
- Constructor functions (of a history) 132
- Contraction (rule of inference) 128
- controller.events** 423

Conventional notation 421
 Cook, Jeff 6, 12
 Cooper, Topher xviii
 Cooperation 252
CORRESPONDING-LITATOM 459
CORRESPONDING-NEGATIVEP 458
CORRESPONDING-NUMBERP 458
CORRESPONDING-NUMBERPS 459
 Cost 54
COUNT 29, 135, 477, 479, 480, 482
COUNT (quantifier operation) 62, 177
 Cowles, John xxiv, 6, 7, 423, 426
 Crawford, Jimi xviii
 Cross-fertilization 212, 270
cstring.events 432
 Current frame 307
 Cut (rule of inference) 128

D

DARPA xix, xxv
 Data base 221, 232, 319, 350, 352
 Data structures 36, 44
 Data types 22, 131
DATA-BASE 225, 319
DCL 224, 323
 Decimal notation 30
 Default path name 343
 Default value 23, 28
DEFAULT-NQTHM-PATH 343
 Definition principle 40, 50, 180, 324, 393
DEFN 224, 324
DEFN-SK xxiii, 435
DEFTHEORY 224, 326
DELETE 79, 468
 Delimited by #| and |# 144
 Delta2 422
DENOMINATOR-SEQUENCE 444
DEQUEUE 75
 Destructor elimination 212, 269
 Destructor term 293
 Destructor term restriction 293
DIFFERENCE 51, 166
 Digit character, base n 141
 Digit value 141
 Digital xix
DISABLE 224, 328
DISABLE (PROVE-LEMMA hint) 367
DISABLE-THEORY 224, 330
DISABLE-THEORY (PROVE-LEMMA hint) 367
 Disabled 275
 Display (of a formal term) 124
 Display (of a token tree) 147

DiVito, Ben 6, 8
DO-EVENTS 226, 331
DO-FILE 332
DO-NOT-GENERALIZE (PROVE-LEMMA hint) 370
DO-NOT-INDUCT (PROVE-LEMMA hint) 370
DO-RETURN (quantifier operation) 62, 177
 Dooley, Sam xviii
 Dot notation 32, 158, 227
 Dot token (.) 142
DOT-CRITERION 453
DOT-TOKEN 445
 Dotted token tree 143
DOTTED-PAIR 446
DOTTED-S-EXPRESSION 446
DOUBLE-LIST 96
DOUBLET 465
 Dribble file 227
DUPLICATESP 465

E

Editing 257
 Edwards, Martyn 11
ELE 75
ELIM 222, 269, 293, 333, 412
 Elimination variable 293
 Embedded formal systems 114
EMIT 450
EMPTY 72
EMPTY-PSTK 453
EMPTY-QUEUEP 74
EMPTY-STACK 461
ENABLE 224, 333
ENABLE (PROVE-LEMMA hint) 367
ENABLE-THEORY 224, 333
ENABLE-THEORY (PROVE-LEMMA hint) 367
 Enabled 275
 Encryption 6, 485
 Endorsed (data base) 361
ENQUEUE 75
 Environment 52
 Epsilon-naught 164
EQLN 446
EQUAL 22, 129
 Equality 127, 188
 Equality Axiom (schema) for functions 128
 Equality Axiom (schema) for predicates 128
 Erratic 379
ERROR 334
 Error handling 226, 334
 Escape mechanism 162

- Euclid's theorem 6
- EVAL\$** 52, 57, 174
- Evaluation 52
- Event 223
- Event commands 336
- Event name 351
- EVENT-FORM (DATA-BASE query)** 321
- EVENTS-SINCE** 336
- Example files 5, 423
- examples** 423
- Executable counterpart 337
- EXISTS** (quantifier operation) 62, 177
- EXP** 443
- EXPAND (PROVE-LEMMA hint)** 365
- Expanding (a definition) 281
- Expansion (of backquote) 151
- Expansion (of readmacros) 150
- Expansion (rule of inference) 128
- Explicit value denoted (by descriptor) 159
- Explicit value descriptor 159
- Explicit value escape descriptor 162
- Explicit value guts 337
- Explicit value preserving 338
- Explicit value term 137, 339
- EXPLICIT-VALUE-DESCRIPTOR** 464
- EXPLODE** 63
- EXPLODED-LITATOM** 473
- Explosion 154
- EXPLOSION** 459
- expr-compiler.events** 428
- EXSYN** 472
- Extended syntax 30, 138, 164, 178, 227, 437
- Extension (of file name) 339, 340
- Extension principles 120
- extras.events** 430
-
- F
- F** 22, 127, 129, 155
- F91** 95
- Factorial 6
- FAILED-EVENTS** 339
- FALSE** 22, 129
- *1*FALSE** 159, 374
- False branch (of an **IF**) 22
- FALSEP** 22, 130
- FATAL ERROR** 334
- FCONS** 458
- Fermat's theorem 6
- Fertilize 270
- fib2.events** 430
- Fibonacci numbers 6
- fibsums.events** 423
- File 339
- File name 339
- FINAL-CDR** 53
- First-level function application 293
- FIRST-N** 443
- FIX** 51, 136
- FIX-COST** 170
- fixnum-gcd.events** 432
- Flag 46
- Flag file protocol 358
- Flatau, Art xxiv, 6, 12, 426
- FLATTEN** 45
- Flawed (induction) 217
- FLIST1** 459
- FLIST2** 459
- FLIST3** 459
- FLIST4** 459
- FLIST5** 459
- FLIST6** 459
- FLIST7** 459
- Floyd, Bob 114
- FM8501 11
- fm8501.events** 428
- FM8502 11, 12
- FM9001 11, 12, 427
- fm9001-replay.events** 427
- fmax.events** 433
- FN-NEST** 459
- Folding 281
- Foldr 8
- foldr.events** 428
- FOR** 61, 96, 177, 178
- FOR** expression denoted 178
- FOR-OPERATIONP** 466
- Formal grammar 140
- Formal metatheory 168
- Formal syntax 122
- Formalization 69, 70
- Formals 40
- FORMALS** 53, 57, 174, 175, 176
- Formula 124, 130, 264
- FORMULA (DATA-BASE query)** 322
- Fortran 10, 114
- fortran.events** 424, 427
- Frame 307
- Free variables (in functional substitutions) 184
- Free variables (in hypotheses) 280, 404
- Free variables (in **LAMBDA** expressions) 184
- FREE-VARS** 101
- FROM-TO** 43
- fs-examples.events** 424
- fsrch.events** 427
- Full trace 372
- FULL-QUEUEP** 74

FULL-TRACE (R-LOOP special form) 375
 Function symbol 20, 122, 351
 Functional instantiation 185
 Functional instantiation (of a formula) 183
 Functional instantiation (of a term) 183
 Functional semantics 8
 Functional substitution 183, 346
FUNCTIONALLY-INSTANTIATE 224, 344

G

Game playing 10
 Garbage collection 12
 Gauss's law 6, 485
gauss.events 424
 Gcd algorithm 11
gcd.events 433
gcd3.events 433
 GCL xxii, 414
 Generalization 7, 212, 261, 271
GENERALIZE 222, 271, 294, 346
generalize-all.events 428
GEQ 51, 166
 German, Steve 6, 11
 Glick, Norm xxv
 Global variable 40, 41
 Gloess, Paul 6, 8
 Gnu Common Lisp xxii, 414
 Gnu General Public License 413
 Gödel's incompleteness theorem 7, 118, 485
goedel.events 431
 Goldschlag, David xviii, xxii, 6
 Good, Don xix, xxv, 12
 Governing terms 180
 GPL 413
 Grafton, Bob xix
GREATERP 51, 166
 Green, Mike xxiv, 6, 115, 423
 Gritch (') 142
 Ground units 280
 Ground Zero 19, 20, 122, 303
group.events 433

H

Hagiya, Masami 414
 Halting problem 7, 485
HANDS-OFF (PROVE-LEMMA hint) 369
 Hardware verification 11, 117, 485, 486
 Hexadecimal notation 31, 142
 Higher-order variables 96
 Hints (to **DEFN**) 325
 Hints (to **PROVE-LEMMA**) 364
 History 126

Hoare, Tony 11, 114, 434
 Huang, C.-H. 6, 9
 Huang, C.H. xviii
 Huet, Gerard 400, 423
 Hunt, Warren xviii, xxiv, 6, 11, 103, 117, 427, 428, 485, 486
 Hypotheses (of a rule) 279

I

IBM xix
IDENTITY 167, 278, 366
IF 22, 129
IFF 51, 165
 Immediate dependency 320
IMMEDIATE-DEPENDENTS (DATA-BASE query) 322
IMMEDIATE-SUPPORTERS (DATA-BASE query) 323
IMPLIES 22, 130
IN (FOR keyword) 62, 178
 Incompatible libraries 353
INDUCT (PROVE-LEMMA hint) 369
induct.events 429
 Induction (mechanization) 212, 259, 272
 Induction (rule of inference) 179
 Induction hypothesis 180
 Induction principle 50
 Induction step 180
 Induction variables 180
 Inductive assertions 8
 Infinite loops (in simplifier) 400
 Infinities 86
 Infix 422
 Infix notation 421
INSERT 76
 Installation 413
 Instantiate 129
 Instantiation (rule of inference) 129
 Institute for Computing Science and Computer Applications xix
INTEGERP 445
integers.events 430
 Interaction 257
 Interlisp 61
 Interpreter 52, 168
 Interpreters 104
 Interrupting commands 295
intro-eg.events 426
 Ireland, Terry xxv
 Irrelevancy 212, 272
isqrt-ada.events 433
isqrt.events 427, 433

J

Jamsek, Damir 429
Justification (of an induction) 180

K

Kamp, Hans xviii
Kaufmann, Matt xxii, xxiii, xxiv, 6, 7, 11,
292, 427, 428, 429, 430, 431, 435
KCL 414
Keenan, Tom xix
Kim, Myung xviii, 268
kit.events 425
Knuth, D.E. 400
Knuth-Bendix problem 406
Koenig's tree lemma 7
koenig.events 428
Kolbly, Bill xxiv
Kunen, Ken xxiv, 6, 429, 430
Kyoto Common Lisp 414

L

Lambda calculus 118
LAMBDA expression 183
Last 422
LAST 444
LaTeX 421
Left-hand side 279
LEGAL-CHAR-CODE-SEQ 460
Legato, Bill xxiv, xxv
LEMMA 348
LEN 41, 60
Lengauer, Chris xviii, 6, 9
Length xxiv
LENGTH 43
Length 421
LENGTH 443
LEQ 51, 166
Less than 164
LESSP 50, 51, 164
LET 32, 127, 156
Letovsky, Stan xxiv, 6
LEX2 303, 325
LEX3 303, 325
Lexeme terminating character 145
LEXEMES 450
Lexicographic 50
Library 232, 350, 352
Library incompatibility 353
Linear arithmetic 286
Linear arithmetic rule 277, 286
Link-assembler verification 12

Lisp 4, 33, 34, 44, 53, 61, 119, 121, 159, 227,
228, 257, 295, 335, 337, 339, 371, 437
LIST 33, 127, 156
List concatenation 167
List processing 6
LIST* 31, 127, 157
LIST1 442
LIST2 442
LIST3 442
LISTP 26, 136, 478
Lists 136, 478
LITATOM 26, 33, 136, 479
LITATOM corresponding to **s** 154
Literal (of a clause) 264
Literal atom 33, 37, 136, 193, 479
LOAD-NQTHM 417
Loading 226, 231, 317, 341, 343, 417
locking.events 428
log2.events 433
Logic 4, 19, 119

M

Machine code 11
Maguire, Gerald xxv
MAINTAIN-REWRITE-PATH 349
Majority vote 10
MAKE-ALIST 468
MAKE-LIB 226, 350
MAKE-QUEUE 74
Manna, Zohar 9
Mapping functions 177
Martin, Norman xviii
Matijasevich's lemma 6
MAX 51, 166
MAX (quantifier operation) 62, 177
mc20-0.events 433
mc20-1.events 434
mc20-2.events 434
McCarthy, John 4, 114
McCune, Bill 6
Measure 180
Measured subset 273
MEMBER 52, 167
memchr.events 432
memcmp.events 432
memcpy.events 432
memmove.events 432
memset.events 432
MERGE 102
Merge sort 6
mergesort-demo.events 428
Merging (inductions) 217
Message flow modulator 87

META 222, 267, 289, 351
 Metafunction 212, 267, 289
 Metatheoretic extensibility 267, 289
 Metatheory 168
 Metavariables 125
 Micro-Gypsy 12
 Microcode 11, 485, 486
MINUS 27, 137, 481
 Misra, Jay 9
mjrty.events 428, 434
 Monus 51
 Motorola MC68020 11
MULTIPLY (quantifier operation) 62, 177
mutex-atomic.events 431
mutex-molecular.events 431
mutilated-checkerboard.events 431
 Mutual exclusion 9
 Mutual recursion 40, 46

N
 Nagayama, Misao xxiv, 6, 9, 431
 Nagle, John xviii
 Name 351
 Name (of an event) 223, 224
 NASA xix
 National Computer Security Center xix
 National Science Foundation xix
 National Security Agency xxv
 Natural number (used as term) 30, 155
 Natural numbers 135, 476
naturals.events 430
 NDL 11
 Negative integer (used as term) 30, 155
 Negative integers 137, 481
NEGATIVE-GUTS 27, 137, 481
NEGATIVEP 27, 137, 481
NEGATIVEP corresponding to **n** 153
fn nest around **b** for **s** 131
 New (symbol) 127
new-gauss.events 424
new-prime.events 429
 Newline 145
 Newton's method 10
NIL 33, 127, 155
 Nim 10, 115
nim-piton.events 427
nim.events 430
 Ninety-one function 95
NLISTP 52, 167
NO-BUILT-IN-ARITH (**PROVE-LEMMA** hint) 370
 Non-SUBRP axiom 176, 179, 182

Nondeterminism 102
NONE-OF 132
 Nonprimitive recursive functions 7
 Nonrecursive 42
NOT 22, 130
 Notation, conventional 422
note-100.events 428
NOTE-LIB 226, 352
NQTHM mode 303, 353
 NSA xix
 Number (used as term) 30, 155
 Number theory 6, 25, 51, 83, 135, 166, 200, 270, 476
NUMBER-SIGN-SEQUENCE 444
NUMBERP 25, 135, 476
NUMBERP corresponding to **n** 153
NUMERATOR-SEQUENCE 443
 Numeric sequence 142
NUMERIC-SEQUENCE 444
NUMERIC-VALUE 444
NUMERIC-WORD 445
NUMERIC-WORD-VALUE 445

O

ω 49, 265
 Obvious (simplification) 218
 Octal notation 30, 142
 Office of Naval Research xix, xxv
OK (R-LOOP special form) 375
 Olmstead, Sandy xix, xxv
 Olphie, Ron xxv
ONE-OF 132
OPEN-PAREN 448
 Opening (a definition) 281
 Operating system verification 12
 Operational semantics 8, 104
 Oppen, Derek 400
 Optionally signed base *n* digit sequence 141
OPTIONALLY-SIGNED-BASE-N-DIGIT-SEQUENCE 443
OR 22, 31, 130
 Oracle 103
ORD-LESSP 50, 165
 Ordered lists 78
 Ordered pairs 136, 478
ORDEREDP 78
ORDINALP 50, 165
 Ordinals 49, 164
 Organization (of handbook) 14
 Organization (of theorem prover) 211
 Output abbreviation mode 374
 Overspecification 108

P

PACK 26, 33, 136, 479
PAIRLIST 52, 168
paris-harrington.events 430
PARSE 454
 Parser 437
parser.events 424
 Partial functions 7, 92
 Partial trace 372
partial.events 429
 Path 307
 Pc-Nqthm xxii
 Pecchiari, Paolo xxiv, 6, 430
PERM 79
 Permutation 6
permutationp-subbagp.events 429
 Permutations 78
 Permutative rules 280, 401
 Persistence 310
peter.events 424
 Pierre, Laurence 6, 11
 Pigeon hole principle 6
 Pinsky, Sylvan xxiv
 Piton 12, 486
piton.events 427
 Pixley, Carl xviii
PLUS 31, 51, 136
 Pnueli, Amir 9
 Pool 212
POP 72, 461
POP-PSTK 453
POSITION 443
PPE 353
pr.events 424
 Preferred quotation 170
 Preferred quotation list 170
 Prettyprint 353
 Primary 320
PRIMARY (DATA-BASE query) 321
 Prime 83
PRIME 84
 Prime factorization 6, 83, 485
PRIME-FACTORIZATION 84
PRIME-LIST 85
PRIME1 84
 Print name 27
 Process 212
PRODUCT 85
 Programming languages 104
PROPERP 92, 236
 Propositional axiom 128
 Propositional calculus 127, 188
 Propositional functions 130
PROTO-QUEUE 73

PROTO-QUEUEP 73
PROVE 354
PROVE-FILE 226, 356
PROVE-FILE-OUT 362
PROVE-LEMMA 224, 363
PROVEALL 226, 354
proveall.events 424
 Proved directly 126
 Pruning 311
 Public key encryption 6, 485
 Pure Lisp 7
PUSH 72, 461
 Push down stacks 72
PUSH-PSTK 453

Q

QLIST 73
QMAX 73
qsort.events 434
QT 465
 Quadratic reciprocity 6, 485
quant.events 424
 Quantification 61, 80, 177
 Quantifier manipulation 96
QUANTIFIER-INITIAL-VALUE 177
QUANTIFIER-OPERATION 177
QUEUEP 74
 Queues 73
 Quick sort 11, 434
 Quotation 168, 170
 Quotation list 169
QUOTE 32, 127, 155
***1*QUOTE** 159, 162, 374
QUOTE notation 32, 158, 227
QUOTIENT 51, 166

R

R 376
R-LOOP 225, 371
 Ramsdell, John 6, 7
 Ramsey's theorem 6
ramsey.events 429
 Read, Simon xxiv, 6, 11
READ-TOKEN-TREE 454
 Readable 149
READABLE 455
 Readmacro expansion 150
READMACRO-EXPANSION 456
 Readtable 301
 Real time control 10, 115
 Recognizer 23, 28, 132
 Recognizer functions (of a history) 132

Recursive 42
 Recursive unsolvability 7, 485
REDUCE-TERM-CLOCK 376
 Reflexivity Axiom 128
 Relieving hypotheses 280
REMAINDER 51, 166
 Renaming 185
 Replacement rule 277, 279
 Restoring data base 352
REV1 394, 450
REVERSE 234
REWRITE 222, 259, 266, 277, 378, 395
 Rewrite path 307
 Rewrite strategy 396
REWRITE-APPLY\$ 378
REWRITE-APPLY-SUBR 378
REWRITE-CAR-V&C\$ 379
REWRITE-CAR-V&C-APPLY\$ 379
REWRITE-EVAL\$ 380
REWRITE-V&C\$ 380
REWRITE-V&C-APPLY\$ 380
 Rewriting 266
 Right-hand side 279
 Ripple carry adder 11
 Root name 339, 381
rotate.events 429
 Rotation 6
rpn.events 429
 RSA encryption 6, 485
rsa.events 425
 Rule 222, 275
 Rule name 351
RUS 54
RUSSELL 95
 Russinoff, David xviii, xxiv, 6, 11, 12, 424, 425, 485

S

s 376
 S-expression 150
S-EXPRESSION 455
 Safety properties 8
 Sandel, Charles xxv
SATELLITES (DATA-BASE query) 322
SATISFIES 462
 Satisfying (a type restriction) 132
 Saving 226
 Saving data base 350
scheduler.events 431
 Schelter, Bill xviii, xxii, 414, 418
 Schemas 96
 Scherlis, Bill xxv
 Science Research Council xix

Scribe 421
 Semantics of programming languages 104
 Separating comment 145
 Separation (suitable) 146
SET-STATUS 224, 381
SET-TIME-LIMIT 385
SETQ (R-LOOP special form) 375
 Shankar, N. 7
 Shankar, Natarajan xviii, xxiv, 6, 118, 188, 431, 485
 Shell principle 22, 28, 72, 73, 131, 133, 298, 475
SHELL-BASE-FUNCTION 461
SHELL-CONSTRUCTOR-FUNCTION 461
SHELL-TYPE-RESTRICTIONS 462
shell.events 426
 Shoenfield's logic 7
 Shoenfield, J.R. 127, 128
 Short stack 12
 Shuffle 6
shuffle.events 429
 Siebert, Ann 6, 12
SIGMA 58
 Simplification 212, 259, 266
 Simulation 10, 115
 Single quote token (') 142
 Single quote token tree 143
SINGLE-QUOTE-TOKEN 445
SINGLE-QUOTE-TOKEN-TREE 447
SINGLETON 446
 Size 29
SKIM-FILE 385
SKIP-PAST-BALANCING-VERTICAL-BAR-NUMBER-SIGN 449
SKIP-PAST-NEWLINE 448
 Skolemization 82
small-machine.events 425
 Smith, Larry xxiv, 6, 12
 Smith, Mike xxiv, xxv, 6
 Some (\exists) xxiii, 80, 435
SORT 76
 Sort 484
 Sorting 76, 78
 Sorting networks 9
 Source files 413
 Space and Naval Warfare Systems Command xix
 Sparc 414
SPECIAL-TOKEN 445
SPECIAL-TOKEN-TREE 447
 Specification problem 14
 Splice escape token tree 144
SPICE-ESCAPE-TOKEN-TREE 447
 Square root 10

- SRI International xix
 - Stack overflow 312, 400
 - STACKP** 72, 461
 - Stacks 72
 - Standard alist 175
 - STANDARD-ALIST** 468
 - STAR-ONE-STAR** 459
 - State-seqp 422
 - STATUS (DATA-BASE query)** 322
 - Status (of a rule) 275
 - STOP** 454
 - strcat.events** 432
 - strchr.events** 432
 - strcmp.events** 432
 - strcoll.events** 432
 - strcpy.events** 432
 - strcspn.events** 432
 - Strictly to the right 144
 - String library 432
 - String searching 10
 - STRIP-CADRS** 465
 - STRIP-CARS** 170
 - strlen.events** 432
 - strncat.events** 432
 - strncmp.events** 432
 - strncpy.events** 432
 - strpbrk.events** 432
 - strrchr.events** 432
 - strspn.events** 432
 - strstr.events** 432
 - strtok.events** 432
 - Structural induction 205
 - strxfm.events** 432
 - SUB1** 25, 135, 476
 - SUBLIS-VAR** 466
 - Subramanian, Sakthi xxiv, 6, 10, 12, 431
 - SUBRP** 53, 56, 135, 174, 175, 176, 179, 477, 479, 480, 482
 - SUBRP** axiom 135, 175, 179
 - SUBSETP** 445
 - SUBST** 48
 - Substitute 129
 - Substitution 129
 - Substitution of equals for equals 191
 - Substring delimited by #| and |# 144
 - Subterm 125
 - Subtraction 166
 - Suggested (induction) 217
 - Suitable separation 146
 - Suitable trailer 146
 - SUM** (quantifier operation) 62, 177
 - SUM-CDRS** 170
 - Super-tame 291
 - switch.events** 434
 - Symbol 122
 - SYMBOLP** 460
 - SYMBOLPS** 466
 - Syntax 20, 62, 122, 138, 164, 178, 227, 437
- T**
- T** 22, 127, 129, 155
 - Takeuchi function 7
 - Talcott, Carolyn xxiv, 6, 9, 426, 431
 - Tame term 290
 - Target function symbol 289
 - Task isolation 12
 - Tautology checking 7, 485
 - Tautology theorem 7
 - tautology.events** 431
 - Teitelman, Warren 61
 - Term 20, 124
 - Term (used as formula) 130
 - Terminating character 145
 - Termination 92
 - Test (of an **IF**) 22
 - TEST-READER** 457
 - Text editing 257
 - Theorem 127
 - Theory 326, 330, 333, 367, 381
 - Theory name 326, 381
 - THM** mode 303, 386
 - Tic-tac-toe 10, 115
 - tic-tac-toe.events** 425
 - Tice, Laura xxv
 - Time 422
 - Time limits 385
 - Time triple 386
 - TIMES** 31, 51, 166
 - tmi.events** 425
 - To the right (strictly) 144
 - TOGGLE** 224, 387
 - TOGGLE-DEFINED-FUNCTIONS** 224, 387
 - Token tree 143
 - TOKEN-TREE** 446
 - Tolerable functional substitution 184
 - Too many arguments 31
 - TOP** 72, 461
 - Top function symbol 125
 - TOP-PSTK** 453
 - tossing.events** 431
 - Total 291
 - Towers of Hanoi 116
 - TRACE (R-LOOP special form)** 375
 - Trace mode 372
 - Trailer (suitable) 146
 - Train-entry 422
 - Train-entry, defined 422

train.events 431
 Trains 422
 Transactional data base model 6
TRANSLATE 388, 469
 Translation (of an s-expression) 155
 Tree 39, 136, 478
 Trivial (simplification) 218
TRUE 22, 129
***1*TRUE** 159, 374
 True branch (of an **IF**) 22
TRUEP 22, 130
 Turing completeness 7
 Twos-complement 11
 Type (of a constructor or base) 132
 Type prescription formula 283
 Type prescription rule 277, 282
 Type restriction 23, 28, 132, 137
 Type restriction term 132
 Type set 265
 Type set described (by a term) 282
 Type set term 282

U

UBT 225, 388
UNABBREV (R-LOOP special form) 375
UNBREAK-LEMMA 389
 Undefined function 323
UNDO-BACK-THROUGH 225, 390
UNDO-NAME 225, 390
 Undotted token tree 143
UNION 52, 167
UNION (quantifier operation) 62, 177
 Unity 9
 University of Texas at Austin xvii
UNPACK 26, 136, 479
unsolv.events 425
UNTRACE (R-LOOP special form) 375
UPCASE 450
 Update-state 422
UPPER-DIGITS 442
USE (PROVE-LEMMA hint) 364

V

V&C\$ 52, 57, 92, 168, 171
V&C-APPLY\$ 57, 173
 Value (and cost) 54
 Value, of a digit 141
 Value, of a numeric sequence 142
 Variable symbol 20, 122, 351
 Variables (of a term) 125
 Variant 185
 Verhoeff, Tom xviii

Verification condition generator 10, 114
 Verification conditions 8
 Verkest, Diederik 6, 11
 VHDL 11

W

Wachter, Ralph xix, xxv
 Wagner, Nancy xxv
 Wang algorithm 7, 485
WARNING 334
 Weber-Wulff, Debbie 6, 8
 Well-formed s-expression 155
WHEN (FOR keyword) 62, 178
 White space character 145
WHITE-SPACEP 450
 Wilding, Matt xxii, xxiv, 6, 10, 12, 430, 431
 Wilson's theorem 6
wilson.events 425
 Witness 82
 Witthoft, Morgan xxiv
 Word 143
WORD 445
WORD-CHARACTERS 445
 Wrong number of arguments 31

X

Xerox xix

Y

Yew, P.C. xxv
 Young, Bill xviii, xxiv, 6, 8, 10, 11, 12
 Young, William 431
 Yu, Yuan xxiv, 6, 11, 431, 432, 433, 434
 Yuasa, Taiichi 414

Z

ZERO 25, 135, 476
zero.events 434
ZEROP 51, 136
ztak.events 425

\ (application of a functional substitution) 184

` (backquote) 34, 142
 ` , Nqthm interpretation of 301

|#-occurrence 144

Contents

Series Foreword	xiii
Preface to the First Edition	xv
Preface to the Second Edition	xxi
 Part I. The Logic	
 1. Introduction	3
1.1. A Summary of the Logic and Theorem Prover	4
1.2. Some Interesting Applications	5
1.3. The Organization of this Handbook	14
 2. A Primer for the Logic	19
2.1. Syntax	20
2.2. Boolean Operators	22
2.3. Data Types	22
2.4. Extending the Syntax	30
2.5. Conventional Data Structures	36
2.6. Defining Functions	40
2.7. Recursive Functions on Conventional Data Structures	44
2.8. Ordinals	49
2.9. Useful Functions	50
2.10. The Interpreter	52
2.11. The General Purpose Quantifier	61
2.12. Introducing Functions by Constraint	64
 3. Formalization Within the Logic	69

3.1. Elementary Programming Examples	72
3.2. Elementary Mathematical Relationships	77
3.3. Dealing with Omissions	80
3.4. Dealing with Features	92
3.5. Nondeterminism	102
3.6. Computing Machines and Programming Languages	104
3.7. Embedded Formal Systems	114
4. A Precise Description of the Logic	119
4.1. Apologia	119
4.2. Outline of the Presentation	120
4.3. Formal Syntax	122
4.4. Embedded Propositional Calculus and Equality	127
4.5. The Shell Principle and the Primitive Data Types	131
4.6. Explicit Value Terms	137
4.7. The Extended Syntax	138
4.8. Ordinals	164
4.9. Useful Function Definitions	165
4.10. The Formal Metatheory	168
4.11. Quantification	177
4.12. SUBRPs and non-SUBRPs	179
4.13. Induction and Recursion	179
4.14. Constraints and Functional Instantiation	182
5. Proving Theorems in the Logic	187
5.1. Propositional Calculus with Equality	188
5.2. Theorems about IF and Propositional Functions	191
5.3. Simple Theorems about NIL, CONS, and APPEND	193
5.4. The Associativity of APPEND	196
5.5. Simple Arithmetic	200
5.6. Structural Induction	205
 Part II. Using the System	
6. Mechanized Proofs in the Logic	211
6.1. The Organization of Our Theorem Prover	211
6.2. An Example Proof—Associativity of TIMES	214
6.3. An Explanation of the Proof	217
7. An Introduction to the System	221
7.1. The Data Base of Rules	221
7.2. Logical Events	223
7.3. A Summary of the Commands	224
7.4. Errors	226

7.5. Aborting a Command	226
7.6. Syntax and the User Interface	227
7.7. Confusing Lisp and the Logic	228
8. A Sample Session with the Theorem Prover	231
9. How to Use the Theorem Prover	251
9.1. REVERSE-REVERSE Revisited—Cooperatively	252
9.2. Using Lisp and a Text Editor as the Interface	257
9.3. The Crucial Check Points in a Proof Attempt	259
10. How the Theorem Prover Works	263
10.1. The Representation of Formulas	264
10.2. Type Sets	265
10.3. Simplification	266
10.4. Elimination of Destructors	269
10.5. Heuristic Use of Equalities	270
10.6. Generalization	271
10.7. Elimination of Irrelevancy	272
10.8. Induction	272
11. The Four Classes of Rules Generated from Lemmas	275
11.1. REWRITE	277
11.2. META	289
11.3. ELIM	293
11.4. GENERALIZE	294
12. Reference Guide	295
12.1. Aborting or Interrupting Commands	295
12.2. ACCUMULATED-PERSISTENCE	297
12.3. ADD-AXIOM	298
12.4. ADD-SHELL	298
12.5. AXIOM	300
12.6. BACKQUOTE-SETTING	301
12.7. BOOT-STRAP	303
12.8. BREAK-LEMMA	305
12.9. BREAK-REWRITE	312
12.10. CH	314
12.11. CHRONOLOGY	316
12.12. COMMENT	316
12.13. COMPILE-UNCOMPILED-DEFNS	317
12.14. CONSTRAIN	317
12.15. DATA-BASE	319
12.16. DCL	323
12.17. DEFN	324

12.18. DEFTHEORY	326
12.19. DISABLE	328
12.20. DISABLE-THEORY	330
12.21. DO-EVENTS	331
12.22. DO-FILE	332
12.23. ELIM	333
12.24. ENABLE	333
12.25. ENABLE-THEORY	333
12.26. Errors	334
12.27. Event Commands	336
12.28. EVENTS-SINCE	336
12.29. Executable Counterparts	337
12.30. Explicit Values	339
12.31. Extensions	339
12.32. FAILED-EVENTS	339
12.33. File Names	339
12.34. FUNCTIONALLY-INSTANTIATE	344
12.35. Functional Substitution	346
12.36. GENERALIZE	346
12.37. Hints to DEFN	347
12.38. Hints to PROVE-LEMMA, CONSTRAIN and FUNCTIONALLY-INSTANTIATE	347
12.39. LEMMA	348
12.40. MAINTAIN-REWRITE-PATH	349
12.41. MAKE-LIB	350
12.42. META	351
12.43. Names—Events, Functions, and Variables	351
12.44. NOTE-LIB	352
12.45. NQTHM Mode	353
12.46. PPE	353
12.47. PROVE	354
12.48. PROVEALL	354
12.49. PROVE-FILE	356
12.50. PROVE-FILE-OUT	362
12.51. PROVE-LEMMA	363
12.52. R-LOOP	371
12.53. REDUCE-TERM-CLOCK	376
12.54. REWRITE	378
12.55. REWRITE-APPLY\$	378
12.56. REWRITE-APPLY-SUBR	379
12.57. REWRITE-CAR-V&C\$	379
12.58. REWRITE-CAR-V&C-APPLY\$	380

Contents	xi
12.59. REWRITE-EVAL\$	380
12.60. REWRITE-V&C\$	380
12.61. REWRITE-V&C-APPLY\$	381
12.62. Root Names	381
12.63. Rule Classes	381
12.64. SET-STATUS	381
12.65. SET-TIME-LIMIT	385
12.66. SKIM-FILE	385
12.67. THM Mode	386
12.68. Time Triple	386
12.69. TOGGLE	387
12.70. TOGGLE-DEFINED-FUNCTIONS	387
12.71. TRANSLATE	388
12.72. UBT	388
12.73. UNBREAK-LEMMA	389
12.74. UNDO-BACK-THROUGH	390
12.75. UNDO-NAME	390
13. Hints on Using the Theorem Prover	393
13.1. How to Write “Good” Definitions	393
13.2. How to Use REWRITE Rules	395
13.3. How to Use ELIM Rules	412
14. Installing Nqthm	413
14.1. The Source Files	413
14.2. Compilation	415
14.3. Loading	417
14.4. Executable Images	417
14.5. Installation Problems	418
14.6. Shakedown	420
14.7. Printing Nqthm Events in Conventional Notation	421
14.8. Example Event Files	423
14.9. DEFN-SK	435
Appendix I. A Parser for the Syntax	437
Appendix II. The Primitive Shell Axioms	475
II.1. The Natural Numbers	476
II.2. The Ordered Pairs	478
II.3. The Literal Atoms	479
II.4. The Negative Integers	481

Appendix III. On the Difficulty of Proofs	483
References	493
Index	507